O'REILLY®

TURING 图灵程序设计丛书

Python 深度学习入门

从零构建CNN和RNN

Deep Learning from Scratch



[美] 塞思·韦德曼 著

郑天民 译

 中国工信出版集团  人民邮电出版社 POSTS & TELECOM PRESS

译者简介

郑天民

日本足利工业大学信息工程学硕士，研究方向为人工智能在大规模调度系统中的应用，在国际三大索引上发表多篇论文。拥有十余年软件行业从业经验，曾在多家大型上市公司、互联网电商、健康类独角兽公司担任CTO、系统分析架构师和技术总监等职务。阿里云MVP、腾讯云TVP、TGO鲲鹏会会员。著有《Spring响应式微服务》《系统架构设计》《向技术管理者转型》《微服务设计原理与架构》《微服务架构实战》。

数字版权声明


图灵程序设计丛书

Python深度学习入门：从零构建CNN和RNN

Deep Learning from Scratch

[美] 塞思·韦德曼 著
郑天民 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®
O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Python深度学习入门 ： 从零构建CNN和RNN / (美)
塞思·韦德曼 (Seth Weidman) 著 ; 郑天民译. -- 北京:
人民邮电出版社, 2021.2
（图灵程序设计丛书）
ISBN 978-7-115-55564-9

Ⅰ. ①P… Ⅱ. ①塞… ②郑… Ⅲ. ①软件工具－程序
设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2020)第249292号

内 容 提 要

本书全面介绍了深度学习知识，借助数学公式、示意图和代码，旨在帮助读者从数学层面、概念层面和应用层面理解神经网络。读者可以跟随本书构建和训练神经网络模型，从而解决实际问题。另外，本书着重介绍卷积神经网络和循环神经网络，并提供 PyTorch 开源神经网络库的使用方法，有助于学习构建更高级的神经网络架构。

本书适合软件工程师、数据分析师，以及其他对机器学习和数据科学感兴趣的读者阅读。

◆ 著 [美] 塞思·韦德曼
译 郑天民
责任编辑 谢婷婷
责任印制 周昇亮

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 https://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：13.25
字数：315千字 2021年 2 月第 1 版
印数：1－2 500册 2021年 2 月北京第 1 次印刷

著作权合同登记号 图字：01-2020-0424号

定价：79.00元

读者服务热线：(010)84084456 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

译者序

当下互联网行业飞速发展，快速的业务更新和产品迭代给系统开发技术和模式带来新的挑战。随着业务场景的日益丰富以及业务数据的积累和沉淀，多元化搜索、数据挖掘、自然语言处理、多媒体学习、语音处理、个性化推荐等已经成为当下互联网系统中所必备的技术体系。而在所有这些技术体系的背后，深度学习都发挥着巨大的作用，另外在日常开发过程中的应用也非常广泛。因此，深度学习俨然成为人工智能领域最热门的研究方向。

深度学习作为机器学习的一个分支，近年来得到了长足的发展。深度学习的概念源于人工神经网络的研究，本质上是包含多个隐藏层的多层感知器结构。神经网络是一个复杂的概念，既包含丰富的理论体系，也涉及大量的数学推导工作。如何高效理解和掌握神经网络是深度学习初学者所面临的一大挑战。为此，我们首先需要具备基本的思维模型，并理解神经网络的组成结构以及运行原理。接着，通过从零开始构建深度学习模型，来理解深度学习中各个核心组件的原理和运行效果。然后，有了前面的基础，进一步学习面向特定应用场景的卷积神经网络（CNN）以及循环神经网络（RNN）。最后，通过一款主流的深度学习开发框架来把所掌握的深度学习模型应用到系统开发过程中。这对掌握深度学习技术而言是一种合理的学习方法论。

本书正是基于上述方法论来对深度学习的方方面面展开讨论的，在内容上详细阐述了关于深度学习的以下核心主题。

- 理解深度学习的思维模型和基本原理，对理解神经网络所需的函数、导数、链式法则等基本概念进行讨论和推导，并分析神经网络的基本结构。
- 构建深度学习模型，从零开始构建一款能够运行的深度学习模型，并掌握损失函数、动量、学习率衰减、dropout 等核心组件。
- 构建 CNN 和 RNN，在原有深度学习模型的基础上结合具体业务场景，了解构建这两种典型神经网络的系统方法。
- 使用 PyTorch 实践神经网络，用 PyTorch 这款主流的深度学习开发框架来实现各种神经网络模型。

作者塞思 • 韦德曼是深度学习领域的资深专家，他善于通过简单明了的方式介绍复杂的概念。因此，这本书可以称作深度学习领域的综合性教程。无论从深度还是广度上，全书都对深度学习的概念和实践方法做了全面的介绍，这体现了作者对这些主题的独到见解，读完让人受益匪浅。这本书对知识体系的构建以及细节的把控也让人印象深刻，从基本概念出发，通过丰富而简洁的代码示例，给出这些概念的实现方案。行文上层层递进，娓娓道来，帮忙大家从入门走向精通。更为重要的是，本书不仅介绍了深度学习的各项功能特性，还提供了一系列面向实战的最佳实践，可以作为广大技术人员的开发指南。

由于时间仓促，译者的水平和经验有限，书中难免有欠妥和错误之处，恳请读者批评指正。

郑天民
2020 年 12 月于杭州钱江世纪城

前言

在学习神经网络和深度学习时，你可能会搜索到大量的资源，从博客文章到 MOOC[1]（如 Coursera 和 Udacity 提供的那些在线课程），甚至还有一些书……但是，这些资源的质量参差不齐，我在前几年开始探索这个主题时就已经对此有所了解。既然你开始阅读本书，就说明你之前了解的所有关于神经网络的解释在一定程度上有所欠缺。刚开始学习这一主题时，我有过同样的经历。就像盲人摸象似的，各种各样的解释描述了不同的方面，但都没有提供一个整体的描述。基于这种情况，我便开始编写本书。

现有的神经网络资源主要分为两类。一类资源侧重于概念领域和数学领域，包含两个方面：一方面是用两端有箭头的线连接圆来形成示意图，这种方式在解释神经网络时非常常见；另一方面是用大量数学公式来解释运行机制，这样做有助于“理解理论”。这类资源的一个典型例子是 Ian Goodfellow 等人所著的《深度学习》，这本书非常优秀。

另一类资源则包含密集的代码块，在运行这些代码块时，它们似乎会显示随时间减少的损失值。也就是说，神经网络会“学习”。例如，PyTorch 文档中的以下示例确实定义并训练了一个基于随机生成数据的简单神经网络：

```
# N是批次大小，D_in是输入维度，H是隐藏维度，D_out是输出维度
N, D_in, H, D_out = 64, 1000, 100, 10

# 创建随机输入数据和输出数据
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# 随机初始化权重
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)
learning_rate = 1e-6
for t in range(500):
```

注 1：massive open online course，大规模开放式在线课程。

```
# 前向传递：计算预测值y
h = x.mm(w1)
h_relu = h.clamp(min=0)
y_pred = h_relu.mm(w2)

# 计算并输出损失值
loss = (y_pred - y).pow(2).sum().item()
print(t, loss)

# 反向计算w1和w2相对于损失值的梯度
grad_y_pred = 2.0 * (y_pred - y)
grad_w2 = h_relu.t().mm(grad_y_pred)
grad_h_relu = grad_y_pred.mm(w2.t())
grad_h = grad_h_relu.clone()
grad_h[h < 0] = 0
grad_w1 = x.t().mm(grad_h)

# 使用梯度下降更新权重
w1 -= learning_rate * grad_w1
w2 -= learning_rate * grad_w2
```

当然，像这样的解释并不能让我们深入了解“工作机制”，如基本的数学原理、其中的独立神经网络组件以及它们协同工作的方式，等等[2]。

怎么样才能更好地解释神经网络呢？对于这个问题，可以参考关于其他计算机科学概念的解释。如果你想学习排序算法，那么会发现一些教科书中包含以下内容。

- 用简单的语言解释算法。
- 关于算法工作原理的可视化解释，类似于编码面试时在白板上画的那种形式。
- 关于“算法运行机制”的一些数学方面的解释[3]。
- 实现算法的伪代码。

尽管我认为理应用这种方式对神经网络展开介绍，但很少有人（甚至从未有过）会全面地解释神经网络的这些内容，本书旨在填补这一空白。

理解神经网络需要多种思维模型

虽然不是专业的研究人员，也没有取得博士学位，但是我曾经熟练地教授数据科学：我曾与 Metis 公司合作辅导过几个数据科学训练营，并在接下来的一年中走访世界各地，为许多不同行业的公司举办了为期 1 到 5 天的研讨会，向那里的员工解释机器学习和软件工程的基本概念。我一直热爱教学，致力于解决如何最好地解释技术概念这一问题。近年来，我重点关注机器学习和统计学中的概念。对于神经网络，我发现最具挑战性的部分是，为

注 2：这个示例旨在为那些已经了解神经网络的人提供 PyTorch 库的说明，而不是作为指导性教程。尽管如此，许多教程仍遵循这种风格，即仅展示代码和一些简短的注释。

注 3：以排序算法为例，这方面的解释是指为什么该算法能生成正确排序的列表。

“什么是神经网络”传达正确的“思维模型”。这主要是因为，了解神经网络需要的不是一个而是多个思维模型，每一个都说明了神经网络工作方式的不同方面（而且每个方面都必不可少）。为了说明这一点，来看一个例子。以下 4 个句子都是“什么是神经网络”这一问题的正确答案。

- 神经网络是一种数学函数，它接受输入并产生输出。
- 神经网络是多维数组流经的计算图。
- 神经网络由层组成，每层都可以被认为具有许多“神经元”。
- 神经网络是一种通用函数逼近器，从理论上讲可以代表任何监督学习问题的解决方案。

事实上，很多人可能已经听过其中一个或多个解释，并且可能对神经网络的工作原理和含义有一定的了解。但是，要完全理解它们，必须了解它们的所有内容并展示它们之间的关系。例如，神经网络如何表示为与“层”的概念相关联的计算图？此外，为了使所有这些解释更加精确，我们将通过 Python 从零开始实现所有这些概念，并将它们融合在一起，形成可以在笔记本计算机上训练的有效神经网络。尽管我们会在实现细节上花费大量时间，**但是通过 Python 实现神经网络模型的目的是巩固并精确地理解概念，而不是尽可能简洁或高效地编写一个神经网络库**。

本书旨在帮助你充分地理解所有这些思维模型以及它们对神经网络实现方式的影响，这样学习相关概念或在这个领域进一步做项目会变得容易得多。

章节概要

前 3 章是最重要的部分，每一章都可以用一本书来讨论，当然，这里精简了内容。

第 1 章说明如何将数学函数表示为一系列连接在一起构成计算图的运算，并演示如何利用这种表示方法和微积分的链式法则，来计算函数的输出相对于其输入的导数。这一章将介绍一个非常重要的运算，即矩阵乘法，并说明如何让它既能够适应用这种方式表示的数学函数，又可以计算最终进行深度学习所需的导数。

第 2 章直接使用第 1 章中创建的构成要素来构建和训练模型，从而解决实际问题。具体地说，就是使用它们来构建线性回归模型和神经网络模型，并基于真实的数据集预测房价。这一章提出，神经网络比线性回归具有更好的性能，并试图直观地解释原因。在这一章中构建模型所采用的“基本原理”方法，可以帮助你很好地理解神经网络的工作原理，同时也展示了逐步的、纯粹基于基本原理的方法在定义深度学习模型方面的局限性。这便引出了第 3 章的内容。

第 3 章从前两章基于基本原理的方法中提取构成要素，并使用它们来构建构成所有深度学习模型的“更高层次”的组件，即 `Layer` 类、`Optimizer` 类等。这一章的结尾在第 2 章的同一个数据集上训练一个从零定义的深度学习模型，该模型比简单的神经网络具有更好的性能。

事实证明，当基于本书中使用的标准训练技术进行训练时，很少有理论保证具有给定架构的神经网络能够在给定的数据集上找到良好的解决方案。第 4 章介绍最重要的训练技术，这些技术通常会使神经网络更有可能找到好的解决方案。另外，书中会尽可能从数学角度说明它们发挥作用的原理。

第 5 章介绍**卷积神经网络**（convolutional neural network，CNN）背后的基本思想，它是一种专门用于理解图像的神经网络架构。关于 CNN 的解释有很多，本书重点介绍 CNN 的核心要点及其与常规神经网络的区别。比如说，CNN 如何将神经元的每一层都组织成“特征图”，以及如何通过卷积过滤器将其中的两层（每层都由多个特征图组成）连接在一起。此外，就像从零开始对神经网络中的常规层进行编码一样，这一章也将从零开始对卷积层进行编码，加深对其工作原理的理解。

第 1 ～ 5 章构建了一个微型神经网络库，该库将神经网络定义为一系列 `Layer` 类，`Layer` 类本身由一系列 `Operation` 类组成，这些 `Operation` 类向前发送输入，向后发送梯度。实际上，这不是大多数神经网络的实现方式。相反，它们使用一种叫作**自动微分**（automatic differentiation）的技术。第 6 章对自动微分进行简单说明，并用它来引出这一章的核心主题——**循环神经网络**（recurrent neural network，RNN）。这种神经网络架构通常用于理解数据点按顺序出现的数据，例如时间序列数据或自然语言数据。这一章还会解释 vanilla RNN 及其两种变体 GRU 和 LSTM 的工作原理，当然，也会从零开始实现这 3 种神经网络。在整个过程中，你将了解 RNN 及与其变体之间的共同点和不同点。

第 7 章展示如何使用高性能的开源神经网络库 PyTorch，来实现第 1 ～ 6 章中从零开始执行的操作。学习这样的框架对于继续学习神经网络至关重要，但是如果不先深入了解神经网络的工作方式和原理，就接触并学习一个框架，那么从长远来看会严重限制今后的学习。本书各章的目的是让你有能力编写高性能的神经网络（通过教你使用 PyTorch），同时还能让你长期学习并取得成功（在学习 PyTorch 之前先了解基本原理）。最后，这一章将简要说明神经网络如何用于无监督学习。

本书的目标就是成为我在学习神经网络和深度学习时所渴望拥有的那样一本书，希望你从中有所收获。加油！

排版约定

本书采用了以下排版约定。

- **黑体**

  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据类型、环境变量、语句和关键字等。

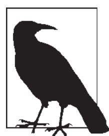
- 等宽粗体（**`constant width bold`**）

  表示应该由用户直接输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。

勾股定理表示为 $a^2 + b^2 = c^2$。



该图标表示一般的注记。

使用示例代码

补充材料（示例代码、练习等）可以从 https://github.com/SethHWeidman/DLFS_code 下载[4]。

本书旨在帮助你完成工作。一般来说，你可以在自己的程序或文档中使用本书提供的示例代码。除非需要复制大量代码，否则无须联系我们获得许可。比如，使用本书中的几个代码片段编写程序无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将本书中的大量示例代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明通常包括书名、作者、出版社和 ISBN，比如"*Deep Learning from Scratch* by Seth Weidman (O'Reilly). Copyright 2019 Seth Weidman, 978-1-492-04141-2"。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly在线学习平台（O'Reilly Online Learning）

O'REILLY®

近 40 年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 https://oreilly.com。

注 4：也可以从图灵社区下载：ituring.cn/book/2759。——编者注

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息[5]。本书的网页是 https://oreil.ly/dl-from-scratch。

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

要更多地了解 O'Reilly 图书、培训课程、会议和新闻，请访问以下网站：http://www.oreilly.com。

我们在 Facebook 的地址如下：http://facebook.com/oreilly。

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia。

致谢

我要感谢编辑 Melissa Potter 和 O'Reilly 团队，他们为本书提供了认真且及时的反馈，并在整个出版过程中详细解答我的疑问。

我要特别感谢一些人，正是他们的努力让机器学习中的技术概念更容易为大多数人所接受，这直接影响了我。我有幸结识了其中一部分人，他们是 Brandon Rohrer、Joel Grus、Jeremy Watt 和 Andrew Trask。

我要感谢我在 Metis 公司和 Facebook 公司的领导，他们十分支持我写作本书。

我要特别感谢 Mat Leonard，在我们决定各自发展之前，他一度参与了本书的写作。Mat 帮助组织了与本书相关的小型图书馆示例 lincoln 的代码，并在前两章的初稿方面给予我非常有用的反馈，同时为这两章撰写了大部分内容。

注 5：也可以通过图灵社区下载示例代码或提交中文版勘误：ituring.cn/book/2759。——编者注

最后，我要感谢我的朋友 Eva 和 John，他俩都直接鼓励并启发我真正开始写作。我还要感谢我在旧金山的许多朋友，他们理解我几个月以来为本书投入了大量的时间，表达了对本书的关心，并且在我需要时坚定地支持我。

电子书

扫描如下二维码，即可购买本书中文版电子版。



第 1 章 基本概念

不要记住这些公式。如果能理解这些概念，那么你就可以自己发明符号。

——John Cochrane，*Investments Notes*

本章旨在解释一些基本的思维模型，这些模型对于理解神经网络的工作原理至关重要。具体地说，本章将介绍**嵌套数学函数**（nested mathematical function）及其**导数**（derivative）。我们将从最简单的构成要素开始逐步研究，证明可以构建由函数链组成的复杂函数。即使其中一个函数是接受多个输入的矩阵乘法，也可以计算函数输出相对于其输入的导数。另外，理解该过程对于理解神经网络至关重要，从第 2 章开始将涉及神经网络的内容。

当围绕神经网络的基本构成要素进行研究时，我们将从 3 个维度系统地描述所引入的每个概念。

- 以一个或多个方程式的形式所表示的数学。
- 解释过程的示意图，类似于在参加编码面试时画在白板上的图表。
- 包含尽可能少的特殊语法的代码（Python 是一个理想选择）。

如前言所述，理解神经网络的一大挑战是它需要多个思维模型。你在本章中就可以体会到这一点：对将讨论的概念来说，以上 3 个维度分别代表不同的基本特征，只有把它们结合在一起，才能对一些概念形成完整的认识，比如嵌套数学函数如何以及为何起作用。注意，要完整地解释神经网络的构成要素，以上 3 个维度缺一不可。

明白了这一点，接下来就可以开始本书的学习了。我将从一些非常简单的构成要素开始讲解，介绍如何基于这 3 个维度来理解不同的概念。第一个构成要素是一个简单但又至关重要的概念：函数。

1.1　函数

什么是函数？如何描述函数？与神经网络一样，函数也可以用多种方法描述，但没有一种方法能完整地描绘它。与其尝试给出简单的一句话描述，不如像盲人摸象那样，依次根据每个维度来了解函数。

数学

下面是两个用数学符号描述的函数示例。

- $f_1(x) = x^2$
- $f_2(x) = \max(x, 0)$

以上有两个函数，分别为 f_1 和 f_2，当输入数字 x 时，第一个函数将其转换为 x^2，第二个函数则将其转换为 $\max(x, 0)$。

示意图

下面是一种描绘函数的方法。

1. 绘制一个 x-y 平面（其中 x 表示横轴，y 表示纵轴）。
2. 绘制一些点，其中，点的 x 坐标是函数在某个范围内的输入（通常是等距的），y 坐标则是该范围内函数的输出。
3. 连接所绘制的点。

这种利用坐标的方法是法国哲学家勒内·笛卡儿发明的，它在许多数学领域非常有用，特别是微积分领域。图 1-1 展示了以上两个函数的示意图。

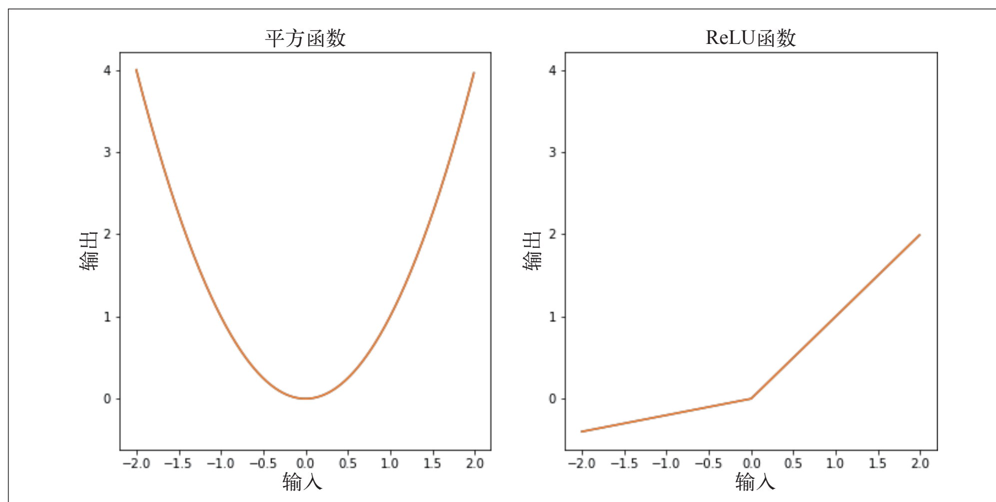


图 1-1：两个连续、基本可微的函数

然而，还有另一种描绘函数的方法，这种方法在学习微积分时并没有那么有用，但是对于思考深度学习模型非常有帮助。可以把函数看作接收数字（输入）并生成数字（输出）的盒子，就像小型工厂一样，它们对输入的处理有自己的内部规则。图 1-2 通过一般规则和具体的输入描绘了以上两个函数。

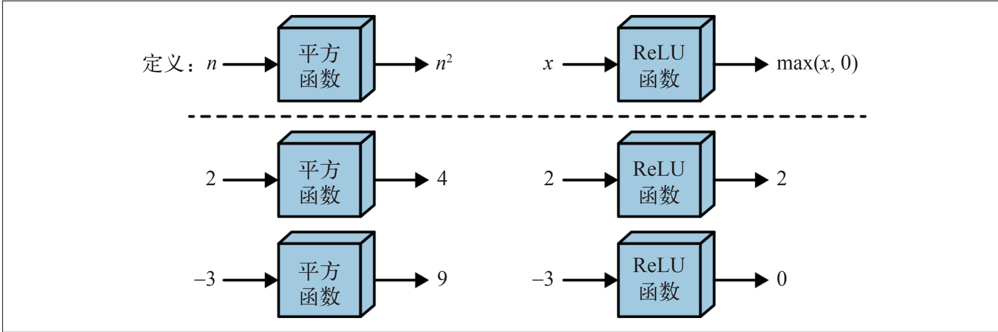


图 1-2：另一种描绘函数的方法

代码

最后，可以使用代码描述这两个函数。在开始之前，先介绍一下 NumPy 这个 Python 库，下面会基于该库编写函数。

1. NumPy 库

NumPy 是一个广泛使用的 Python 库，用于快速数值计算，其内部大部分使用 C 语言编写。简单地说，在神经网络中处理的数据将始终保存在一个**多维数组**中，主要是一维数组、二维数组、三维数组或四维数组，尤其以二维数组或三维数组居多。NumPy 库中的 `ndarray` 类能够让我们以直观且快速的方式计算这些数组。举一个最简单的例子，如果将数据存储在 Python 列表或列表的嵌套列表中，则无法使用常规语法实现数据对位相加或相乘，但 `ndarray` 类可以实现：

```
print("Python list operations:")
a = [1,2,3]
b = [4,5,6]
print("a+b:", a+b)
try:
    print(a*b)
except TypeError:
    print("a*b has no meaning for Python lists")
print()
print("numpy array operations:")
a = np.array([1,2,3])
b = np.array([4,5,6])
print("a+b:", a+b)
print("a*b:", a*b)
```

```
Python list operations:
a+b: [1, 2, 3, 4, 5, 6]
a*b has no meaning for Python lists

numpy array operations:
a+b: [5 7 9]
a*b: [ 4 10 18]
```

ndarray 还具备 n 维数组所具有的多个特性：每个 ndarray 都具有 n 个轴（从 0 开始索引），第一个轴为轴 0，第二个轴为轴 1，以此类推。另外，由于二维 ndarray 较为常见，因此可以将轴 0 视为行，将轴 1 视为列，如图 1-3 所示。

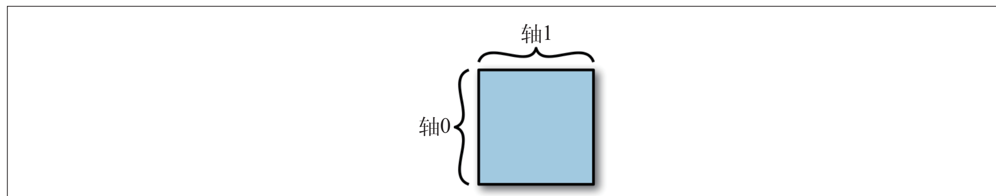


图 1-3：一个二维 ndarray，其中轴 0 为行，轴 1 为列

NumPy 库的 ndarray 类还支持以直观的方式对这些轴应用函数。例如，沿轴 0（二维数组的行）求和本质上就是沿该轴“折叠数组”，返回的数组比原始数组少一个维度。对二维数组来说，这相当于对每一列进行求和：

```
print('a:')
print(a)
print('a.sum(axis=0):', a.sum(axis=0))
print('a.sum(axis=1):', a.sum(axis=1))

a:
[[1 2]
 [3 4]]
a.sum(axis=0): [4 6]
a.sum(axis=1): [3 7]
```

ndarray 类支持将一维数组添加到最后一个轴上。对一个 R 行 C 列的二维数组 a 而言，这意味着可以添加长度为 C 的一维数组 b。NumPy 将以直观的方式进行加法运算，并将元素添加到 a 的每一行中[1]。

```
a = np.array([[1,2,3],
              [4,5,6]])

b = np.array([10,20,30])

print("a+b:\n", a+b)

a+b:
[[11 22 33]
 [14 25 36]]
```

注 1：这样一来，后续便可以轻松地向矩阵乘法添加偏差。

2. 类型检查函数

如前所述，本书代码的主要目标是确保概念描述的准确性和清晰性。随着本书内容的展开，这将变得更具挑战性，后文涉及编写带有许多参数的函数，这些参数是复杂类的一部分。为了解决这个问题，本书将在整个过程中使用带有类型签名的函数。例如，在第 3 章中，我们将使用如下方式初始化神经网络：

```
def __init__(self,
             layers: List[Layer],
             loss: Loss,
             learning_rate: float = 0.01) -> None:
```

仅通过类型签名，就能了解该类的用途。与此相对，考虑以下**可用于定义运算**的类型签名：

```
def operation(x1, x2):
```

这个类型签名本身并没有给出任何提示。只有打印出每个对象的类型，查看对每个对象执行的运算，或者根据名称 x1 和 x2 进行猜测，才能够理解该函数的功能。这里可以改为定义具有类型签名的函数，如下所示：

```
def operation(x1: ndarray, x2: ndarray) -> ndarray:
```

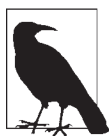
很明显，这是接受两个 ndarray 的函数，可以用某种方式将它们组合在一起，并输出该组合的结果。由于它们读起来更为清楚，因此本书将使用经过类型检查的函数。

3. NumPy 库中的基础函数

了解前面的内容后，现在来编写之前通过 NumPy 库定义的函数：

```
def square(x: ndarray) -> ndarray:
    '''
    将输入ndarray中的每个元素进行平方运算。
    '''
    return np.power(x, 2)

def leaky_relu(x: ndarray) -> ndarray:
    '''
    将Leaky ReLU函数应用于ndarray中的每个元素。
    '''
    return np.maximum(0.2 * x, x)
```



NumPy 库有一个奇怪的地方，那就是可以通过使用 np.*function_name*(ndarray) 或 ndarray.*function_name* 将许多函数应用于 ndarray。例如，前面的 ReLU 函数可以编写成 x.clip(min=0)。本书将尽量保持一致，在整个过程中遵循 np.*function_name*(ndarray) 约定，尤其会避免使用 *ndarray*.T 之类的技巧来转置二维 ndarray，而会使用 np.transpose(*ndarray*, (1, 0))。

如果能通过数学、示意图和代码这 3 个维度来表达相同的基本概念，就说明你已经初步拥有了真正理解深度学习所需的灵活思维。

1.2 导数

像函数一样，导数也是深度学习的一个非常重要的概念，大多数人可能很熟悉。同样，导数也可以用多种方式进行描述。总体来说，函数在某一点上的导数，可以简单地看作函数输出相对于该点输入的“变化率”。接下来基于前面介绍的 3 个维度来了解导数，从而更好地理解导数的原理。

数学

首先来从数学角度精确地定义导数：可以使用一个数字来描述极限，即当改变某个特定的输入值 a 时，函数 f 输出的变化：

$$\frac{\mathrm{d}f}{\mathrm{d}u}(a)=\lim_{\Delta\to 0}\frac{f(a+\Delta)-f(a-\Delta)}{2\times\Delta}$$

通过为 Δ 设置非常小的值（例如 0.001），可以在数值上近似此极限。因此，可以将导数计算为：

$$\frac{\mathrm{d}f}{\mathrm{d}u}(a)=\frac{f(a+0.001)-f(a-0.001)}{0.002}$$

虽然近似准确，但这只是完整导数思维模型的一部分，下面来从示意图的维度认识导数。

示意图

采用一种熟悉的方式：在含有函数 f 图像的笛卡儿坐标系上，简单地画出该函数的一条切线，则函数 f 在点 a 处的导数就是该线在点 a 处的斜率。正如本节中的数学描述一样，这里也可以通过两种方式实际计算这条线的斜率。第一种方式是使用微积分来实际计算极限，第二种方式是在 $a-0.001$ 处和 $a+0.001$ 处取连线 f 的斜率。后者如图 1-4 所示，如果学过微积分，应该会很熟悉。

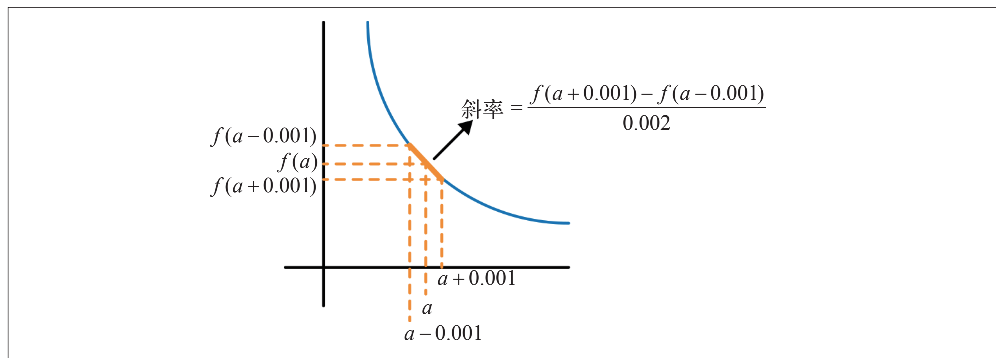


图 1-4：导数即为斜率

正如 1.1 节所述，可以把函数想象成小型工厂。现在想象那些工厂的输入通过一根线连接到输出。求解导数相当于回答这样一个问题：如果将函数的输入 a 拉高一点，或者如果函数在 a 处可能不对称，因此把 a 拉低一点，那么根据工厂的内部运作机制，输出量将以这个小数值的多少倍进行变化呢？如图 1-5 所示。

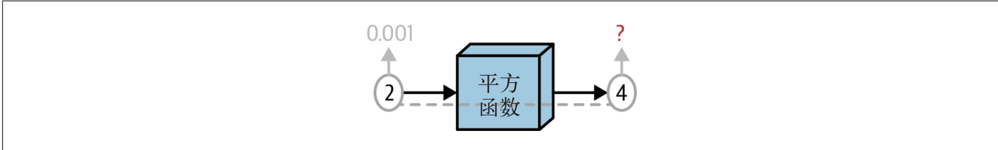


图 1-5：导数可视化的另一种方法

对理解深度学习而言，第二种表示形式比第一种更为重要。

代码

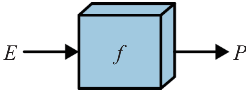
可以通过编码来求解前面看到的导数的近似值：

```
from typing import Callable

def deriv(func: Callable[[ndarray], ndarray],
          input_: ndarray,
          delta: float = 0.001) -> ndarray:
    '''
    计算函数func在input_数组中每个元素处的导数。
    '''
    return (func(input_ + delta) - func(input_ - delta)) / (2 * delta)
```



当说 P 是 E（随机选的字母）的函数时，其实是指存在某个函数 f，使得 $f(E)=P$。或者说，有一个函数 f，它接受对象 E 并产生对象 P。也可以说，P 是函数 f 应用于 E 时产生的任意函数值：



可以将其编码为下面这种形式。

```
def f(input_: ndarray) -> ndarray:
    # 一些转换
    return output

P = f(E)
```

1.3 嵌套函数

现在来介绍一个概念，该概念将成为理解神经网络的基础：函数可以被“嵌套”，从而形成“复合”函数。“嵌套”到底是什么意思呢？假设有两个函数，按照数学惯例，它们分别为 f_1 和 f_2，其中一个函数的输出将成为另一个函数的输入，这样就可以“把它们串在一起”。

数学

嵌套函数在数学上表示为：

$$f_2(f_1(x)) = y$$

这不太直观，因为有个奇怪的地方：嵌套函数是“从外而内”读取的，而而运算实际上是“从内而外”执行的。例如，尽管 $f_2(f_1(x)) = y$ 读作“f_2 接受 f_1 接受 x 对象而产生对象 y”，但其真正含义是“首先将 f_1 应用于 x，然后将 f_2 应用于该结果，最终得到对象 y”。

示意图

要表示嵌套函数，最直观的方法是使用小型工厂表示法，又称盒子表示法。

如图 1-6 所示，输入进入第一个函数，转换之后进行输出。然后，这个输出进入第二个函数并再次转换，得到最终输出。

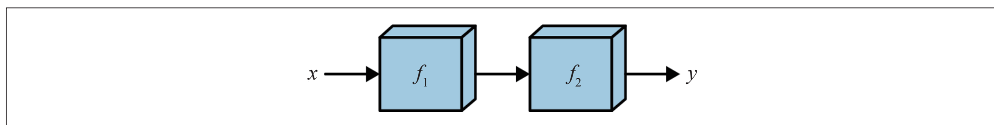


图 1-6：直观地表示嵌套函数

代码

前面已经介绍了两个维度，接下来从代码维度来认识嵌套函数。首先，为嵌套函数定义一个数据类型：

```
from typing import List

# 函数接受一个ndarray作为参数并生成一个ndarray
Array_Function = Callable[[ndarray], ndarray]

# 链是一个函数列表
Chain = List[Array_Function]
```

然后，定义数据如何经过特定长度的链，以长度等于 2 为例，代码如下所示。

```
def chain_length_2(chain: Chain,
                   a: ndarray) -> ndarray:
    '''
    在一行代码中计算“链”中的两个函数。
    '''
    assert len(chain) == 2, \
    "Length of input 'chain' should be 2"

    f1 = chain[0]
    f2 = chain[1]

    return f2(f1(x))
```

另一种示意图

使用盒子表示法描述嵌套函数表明，该复合函数实际上就只是一个函数。因此，可以将该函数简单地表示为 $f_1 f_2$，如图 1-7 所示。

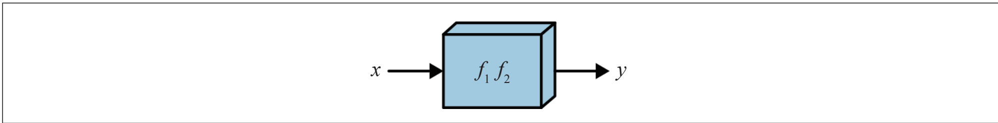


图 1-7：嵌套函数的另一种表示法

此外，在微积分中有一个定理，那就是由“基本可微”的函数组成的复合函数本身就是基本可微的！因此，可以将 $f_1 f_2$ 视为另一个可计算导数的函数。计算复合函数的导数对于训练深度学习模型至关重要。

但是，现在需要一个公式，以便根据各个组成函数的导数来计算此复合函数的导数。这就是接下来要介绍的内容。

1.4　链式法则

链式法则是一个数学定理，用于计算复合函数的导数。从数学上讲，深度学习模型就是复合函数。因此，理解其导数的计算过程对于训练它们非常重要，接下来的几章将详述这一点。

数学

在数学上，这个定理看起来较为复杂，对于给定的值 x，我们有：

$$\frac{\mathrm{d}f_2}{\mathrm{d}u}(x) = \frac{\mathrm{d}f_2}{\mathrm{d}u}(f_1(x)) \times \frac{\mathrm{d}f_1}{\mathrm{d}u}(x)$$

其中 u 只是一个伪变量，代表函数的输入。



当描述具有一个输入和一个输出的函数 f 的导数时，可以将代表该函数导数的函数表示为 $\frac{df}{du}$ 。可以用其他伪变量替代 u，这样做并不会对结果造成影响，就像 $f(x)=x^2$ 和 $f(y)=y^2$ 表示同一个意思一样。

稍后，我们将处理包含**多个**输入（例如 x 和 y）的函数。一旦碰到这种情况，区分 $\frac{df}{dx}$ 和 $\frac{df}{dy}$ 之间的不同含义就是有意义的。

这就是为什么在前面的公式中，我们在所有的导数中将 u 放在了底部：f_1 和 f_2 都是接受一个输入并产生一个输出的函数，在这些情况下（有一个输入和一个输出的函数），我们将在导数符号中使用 u。

示意图

对理解链式法则而言，本节中的数学公式不太直观。对此，盒子表示法会更有帮助。下面通过简单的 f_1f_2 示例来解释导数“应该”是什么，如图 1-8 所示。

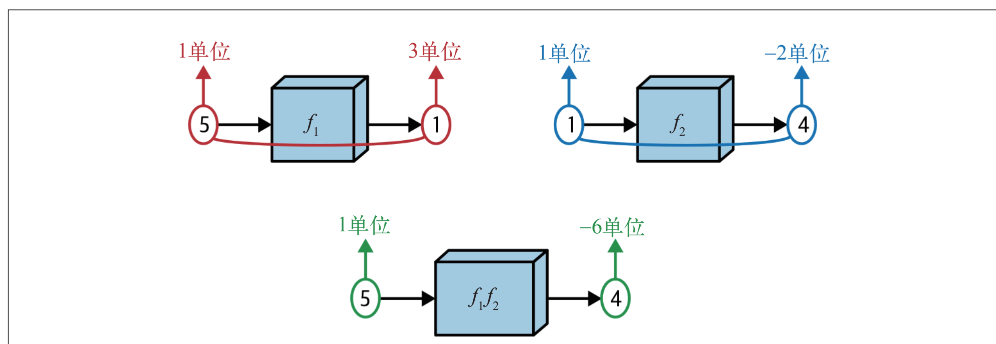


图 1-8：链式法则示意图

直观地说，使用图 1-8 中的示意图，复合函数的导数应该是其组成函数的导数的乘积。假设在第一个函数中输入 5，并且当 $u=5$ 时，第一个函数的导数是 3，那么用公式表示就是 $\frac{df_1}{du}(5)=3$ 。

然后取第一个盒子中的函数值，假设它是 1，即 $f_1(5)=1$ ，再计算第二个函数 f_2 在这个值上的导数，即计算 $\frac{df_2}{du}(1)$ 。如图 1-8 所示，这个值是 –2。

想象这些函数实际上是串在一起的，如果将盒子 2 对应的输入更改 1 单位会导致盒子 2 的输出产生 –2 单位的变化，将盒子 2 对应的输入更改 3 单位则会导致盒子 2 的输出变化 –6（-2×3）单位。这就是为什么在链式法则的公式中，最终结果是一个乘积：$\frac{df_2}{du}(f_1(x))\times\frac{df_1}{du}(x)$ 。

利用数学和示意图这两个维度，我们可以通过使用链式法则来推断嵌套函数的输出相对于其输入的导数值。那么计算这个导数的代码如何编写呢？

代码

下面对此进行编码，并证明按照这种方式计算的导数会产生“看起来正确”的结果。这里将使用 `square` 函数[2]以及 `sigmoid` 函数，后者在深度学习中非常重要：

```
def sigmoid(x: ndarray) -> ndarray:
    '''
    将sigmoid函数应用于输入ndarray中的每个元素。
    '''
    return 1 / (1 + np.exp(-x))
```

现在编写链式法则：

```
def chain_deriv_2(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    使用链式法则计算两个嵌套函数的导数：(f2(f1(x))' = f2'(f1(x))*f1'(x) 。
    '''

    assert len(chain) == 2, \
    "This function requires 'Chain' objects of length 2"

    assert input_range.ndim == 1, \
    "Function requires a 1 dimensional ndarray as input_range"

    f1 = chain[0]
    f2 = chain[1]

    # df1/dx
    f1_of_x = f1(input_range)

    # df1/du
    df1dx = deriv(f1, input_range)

    # df2/du(f1(x))
    df2du = deriv(f2, f1(input_range))

    # 在每一点上将这些量相乘
    return df1dx * df2du
```

图 1-9 绘制了结果，并展示了链式法则的有效性：

```
PLOT_RANGE = np.arange(-3, 3, 0.01)

chain_1 = [square, sigmoid]
chain_2 = [sigmoid, square]
```

注 2：参见 1.1 节的“NumPy 库中的基础函数”部分。

```
plot_chain(chain_1, PLOT_RANGE)
plot_chain_deriv(chain_1,  PLOT_RANGE)

plot_chain(chain_2, PLOT_RANGE)
plot_chain_deriv(chain_2,  PLOT_RANGE)
```

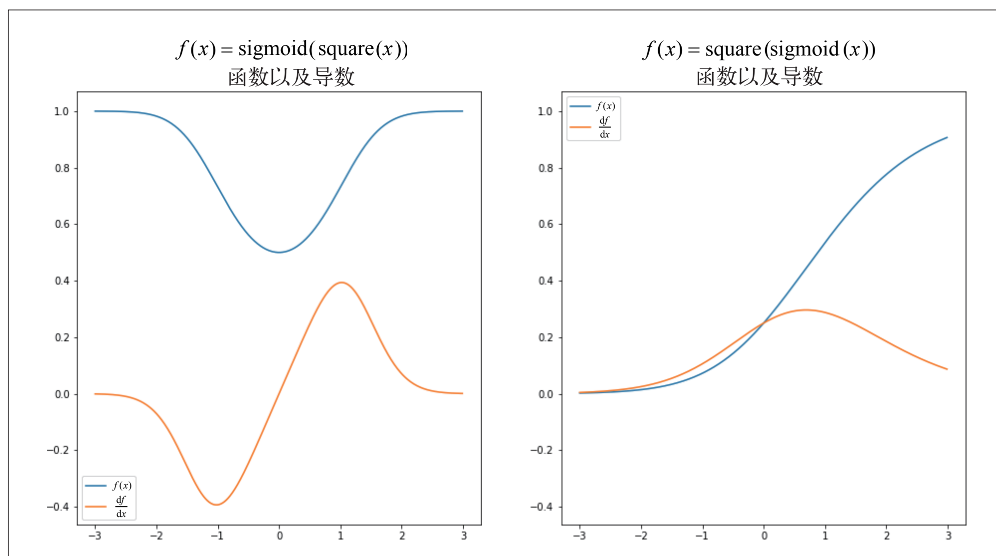


图 1-9：链式法则的有效性[3]

链式法则似乎起作用了。当函数向上倾斜时，导数为正；当函数向下倾斜时，导数为负；当函数未发生倾斜时，导数为零。

因此，实际上只要各个函数本身是基本可微的，就可以通过数学公式和代码计算嵌套函数（或复合函数）的导数，例如 $f_1 f_2$ 。

从数学上讲，深度学习模型是这些基本可微函数的长链。建议花时间手动执行稍长一点的详细示例（参见 1.5 节），这样有助于直观地理解链式法则，包括其运行方式以及在更复杂的模型中的应用。

1.5　示例介绍

仔细研究一条稍长的链，假设有 3 个基本可微的函数，分别是 f_1 、 f_2 和 f_3，如何计算它们的导数呢？从前面提到的微积分定理可以知道，由任意有限个“基本可微”函数组成的复合函数都是基本可微的。因此，计算导数应该不难实现。

注 3：请在图灵社区上查看该图的彩色版本，参见 ituring.cn/book/2759。——编者注

数学

从数学上讲，对于包含 3 个基本可微的函数的复合函数，其导数的计算公式如下：

$$\frac{\mathrm{d}f_3}{\mathrm{d}u}(x)=\frac{\mathrm{d}f_3}{\mathrm{d}u}(f_2(f_1(x)))\times\frac{\mathrm{d}f_2}{\mathrm{d}u}(f_1(x))\times\frac{\mathrm{d}f_1}{\mathrm{d}u}(x)$$

仅仅看公式不是很直观，但相比 1.4 节介绍的 $\frac{\mathrm{d}f_2}{\mathrm{d}u}(x)=\frac{\mathrm{d}f_2}{\mathrm{d}u}(f_1(x))\times\frac{\mathrm{d}f_1}{\mathrm{d}u}(x)$ 适用于长度为 2 的链，两者的基本逻辑是一样的。

示意图

要理解以上公式，最为直观的方法就是通过盒子示意图，如图 1-10 所示。

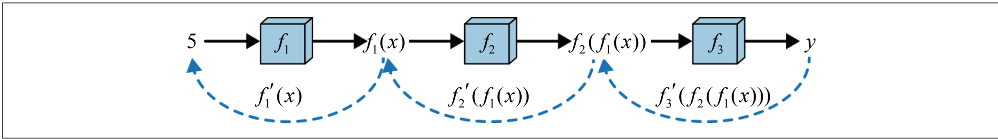


图 1-10：通过盒子表示法理解如何计算 3 个嵌套函数的导数

使用与 1.4 节中类似的逻辑：假设 $f_1 f_2 f_3$ 的输入（称为 a）通过一根线连接到输出（称为 b），则将 a 改变较小量 Δ，将导致 $f_1(a)$ 变化 Δ 的 $\frac{\mathrm{d}f_1}{\mathrm{d}u}(x)$ 倍，进而导致链中的下一步 $f_2(f_1(x))$ 变化 Δ 的 $\frac{\mathrm{d}f_2}{\mathrm{d}u}(f_1(x))\times\frac{\mathrm{d}f_1}{\mathrm{d}u}(x)$ 倍，以此类推，直到最终表示变化的完整公式等于前一个链式法则乘以 Δ。请仔细思考上述解释和图 1-10 中的示意图，无须花费太长时间。在编写代码时，这一点会更容易理解。

代码

在给定组合函数的情况下，如何将本节中的公式转换为计算导数的代码呢？我们可以在这个简单的示例中看到神经网络前向传递和后向传递的雏形：

```
def chain_deriv_3(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    使用链式法则来计算3个嵌套函数的导数：
    (f3(f2(f1)))' = f3'(f2(f1(x))) * f2'(f1(x)) * f1'(x)。
    '''

    assert len(chain) == 3, \
    "This function requires 'Chain' objects to have length 3"

    f1 = chain[0]
    f2 = chain[1]
    f3 = chain[2]
```

```
    # f1(x)
    f1_of_x = f1(input_range)

    # f2(f1(x))
    f2_of_x = f2(f1_of_x)

    # df3du
    df3du = deriv(f3, f2_of_x)

    # df2du
    df2du = deriv(f2, f1_of_x)

    # df1dx
    df1dx = deriv(f1, input_range)

    # 在每一点上将这些量相乘
    return df1dx * df2du * df3du
```

注意，在计算这个嵌套函数的链式法则时，这里对它进行了两次“传递”。

1. “向前”传递它，计算出 f1_of_x 和 f2_of_x，这个过程可以称作（或视作）“前向传递”。
2. “向后”通过函数，使用在前向传递中计算出的量来计算构成导数的量。

最后，将这 3 个量相乘，得到导数。

接下来使用前面定义的 3 个简单函数来说明上面的方法是可行的，这 3 个函数分别为 sigmoid、square 和 leaky_relu。

```
PLOT_RANGE = np.range(-3, 3, 0.01)
plot_chain([leaky_relu, sigmoid, square], PLOT_RANGE)
plot_chain_deriv([leaky_relu, sigmoid, square], PLOT_RANGE)
```

图 1-11 显示了结果。

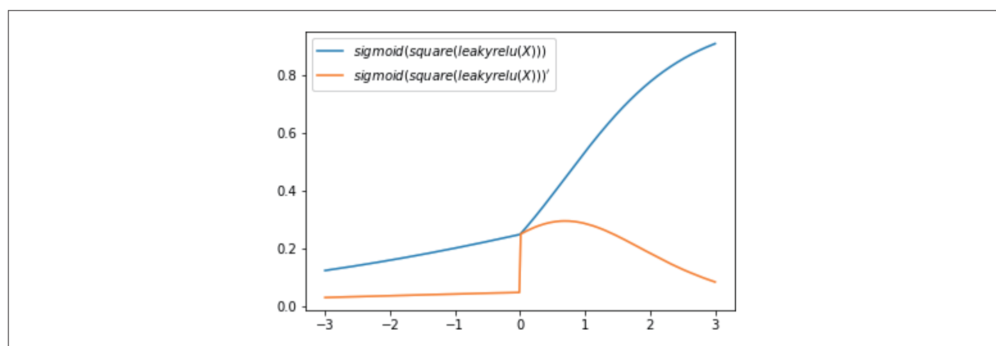


图 1-11：即使使用三重嵌套函数，链式法则也有效[4]

注 4：请在图灵社区上查看该图的彩色版本，参见 ituring.cn/book/2759。——编者注

再次将导数图与原始函数的斜率进行比较，可以看到链式法则确实正确地计算了导数。

基于以上理解，现在再来看一下具有多个输入的复合函数，这类函数遵循已经建立的相同原理，最终将更适用于深度学习。

1.6 多输入函数

至此，我们已经从概念上理解了如何将函数串在一起形成复合函数，并且知道了如何将这些函数表示为一系列输入或输出的盒子，以及如何计算这些函数的导数。一方面通过数学公式理解了导数，另一方面通过“向前”组件和“向后”组件，根据传递过程中计算出的量理解了导数。

在深度学习中处理的函数往往并非只有一个输入。相反，它们有多个输入，在某些步骤中，这些输入以相加、相乘或其他方式组合在一起。正如下面介绍的，我们同样可以计算这些函数的输出相对于其输入的导数。现在假设存在一个有多个输入的简单场景，其中两个输入相加，然后再输入给另一个函数。

数学

在这个例子中，从数学意义上开始讨论实际上很有帮助。如果输入是 x 和 y，那么可以认为函数分两步进行。在步骤 1 中，x 和 y 传到了将它们相加的函数。将该函数表示为 α（整个过程使用希腊字母表示函数名），然后将函数的输出表示为 a。从形式上看，这样很容易表示：

$$a = \alpha(x, y) = x + y$$

步骤 2 是将 a 传给某个函数 σ（σ 可以是任意连续函数，例如 `sigmoid` 函数或 `square` 函数，甚至是名称不以 s 开头的函数）。将此函数的输出表示为 s，也就是：

$$s = \sigma(a)$$

将整个函数用 f 表示，可以写作：

$$f(x, y) = \quad (x + y)$$

从数学意义上理解，这样更为简洁，但这实际上是两个按顺序执行的运算，这一点在表达式中较为模糊。为了说明这一点，来看示意图。

示意图

既然谈到了多输入函数，现在来定义我们一直在讨论的一个概念：用箭头表示数学运算顺序的示意图叫作**计算图**（computational graph）。例如，图 1-12 展示了上述函数 f 的计算图。

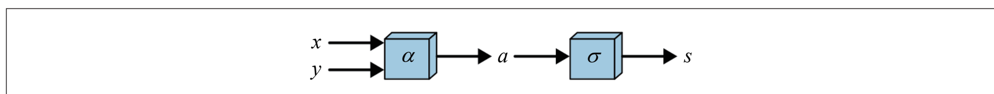


图 1-12：多输入函数

可以看到，两个输入进入 α 输出 a，然后 a 再被传递给了 σ 。

代码

对此进行编码非常简单。但是要注意，必须添加一条额外的断言：

```
def multiple_inputs_add(x: ndarray,
                        y: ndarray,
                        sigma: Array_Function) -> float:
    '''
    具有多个输入和加法的函数，前向传递。
    '''
    assert x.shape == y.shape

    a = x + y
    return sigma(a)
```

与本章前面提到的函数不同，对于输入 ndarray 的每个元素，这个函数不只是简单地进行“逐元素”运算。每当处理将多个 ndarray 作为输入的运算时，都必须检查它们的形状，从而确保满足该运算所需的所有条件。在这里，对于像加法这样的简单运算，只需要检查形状是否相同，以便逐元素进行加法运算。

1.7　多输入函数的导数

可以根据函数的两个输入来计算其输出的导数，这一点很容易理解。

数学

链式法则在这些函数中的应用方式与前面各节介绍的方式相同。由于 $f(x,y)=\sigma(\alpha(x,y))$ 是嵌套函数，因此可以这样计算导数：

$$\frac{\partial f}{\partial x}=\frac{\partial \sigma}{\partial u}(\alpha(x,y))\times\frac{\partial \alpha}{\partial x}(\alpha(x,y))=\frac{\partial \sigma}{\partial u}(x+y)\times\frac{\partial \alpha}{\partial x}(\alpha(x,y))$$

当然，$\frac{\partial f}{\partial y}$ 的计算公式与此相同。

现在要注意：

$$\frac{\partial \alpha}{\partial x}(\alpha(x,y))=1$$

无论 x（或 y）的值如何，x（或 y）每增加一单位，a 都会增加一单位。

基于这一点，稍后可以通过编写代码来计算这样一个函数的导数。

示意图

从概念上讲，计算多输入函数的导数与计算单输入函数的导数所用的方法是相同的：计算每个组成函数“后向”通过计算图的导数，然后将结果相乘即可得出总导数，如图 1-13 所示。

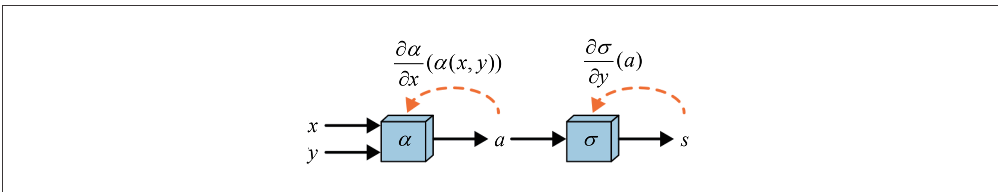


图 1-13：多输入函数后向通过计算图

代码

```
def multiple_inputs_add_backward(x: ndarray,
                                 y: ndarray,
                                 sigma: Array_Function) -> float:
    '''
    计算这个简单函数对两个输入的导数。
    '''
    # 计算前向传递结果
    a = x + y

    # 计算导数
    dsda = deriv(sigma, a)

    dadx, dady = 1, 1

    return dsda * dadx, dsda * dady
```

当然，你可以修改以上代码，比如让 x 和 y 相乘，而不是相加。

接下来将研究一个更复杂的示例，该示例更接近于深度学习的工作原理：一个与前一示例类似的函数，但包含两个向量输入。

1.8 多向量输入函数

深度学习涉及处理输入为**向量**（vector）或**矩阵**（matrix）的函数。这些对象不仅可以进行加法、乘法等运算，还可以通过点积或矩阵乘法进行组合。前面提到的链式法则的数学原

理，以及使用前向传递和后向传递计算函数导数的逻辑在这里仍然适用，本章剩余部分将对此展开介绍。

这些技术将最终成为理解深度学习有效性的关键。深度学习的目标是使模型拟合某些数据。更准确地说，这意味着要找到一个数学函数，以尽可能最优的方式将对数据的**观测**（将作为函数的输入）映射到对数据的**目标预测**（将作为函数的输出）。这些观测值将被编码为矩阵，通常以行作为观测值，每列则作为该观测值的数字特征。第 2 章将对此进行更详细的介绍，现阶段必须能够计算涉及点积和矩阵乘法的复杂函数的导数。

下面从数学维度精确定义上述概念。

数学

在神经网络中，表示单个数据点的典型方法是将 n 个特征列为一行，其中每个特征都只是一个数字，如 x_1、x_2 等表示如下：

$$\boldsymbol{X} = [x_1 \ x_2 \ \cdots \ x_n]$$

这里要记住的一个典型示例是预测房价，第 2 章将从零开始针对这个示例构建神经网络。在该示例中，x_1、x_2 等是房屋的数字特征，例如房屋的占地面积或到学校的距离。

1.9 基于已有特征创建新特征

神经网络中最常见的运算也许就是计算已有特征的**加权和**，加权和可以强化某些特征而弱化其他特征，从而形成一种新特征，但它本身仅仅是旧特征的组合。用数学上的一种简洁的方式表达就是使用该观测值的**点积**（dot product），包含与特征 $(w_1, w_2, \cdots, w_n)$ 等长的一组权重。下面分别从数学、示意图、代码等维度来探讨这个概念。

数学

如果存在如下情况：

$$\boldsymbol{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

那么可以将此运算的输出定义为：

$$N = v(\boldsymbol{X}, \boldsymbol{W}) = \boldsymbol{X} \times \boldsymbol{W} = x_1 \times w_1 + x_2 \times w_2 + \cdots + x_n \times w_n$$

注意，这个运算是**矩阵乘法**的一个特例，它恰好是一个点积，因为 $\boldsymbol{X}$ 只有一行，而 $\boldsymbol{W}$ 只有一列。

接下来介绍用示意图描绘它的几种方法。

示意图

可以通过一种简单的方法来描绘这种运算，如图 1-14 所示。

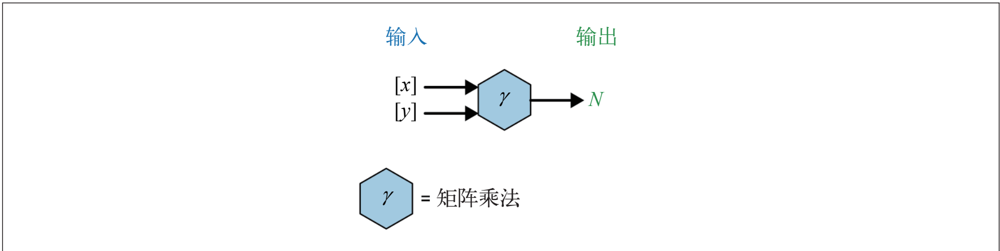


图 1-14：矩阵乘法（向量点积）示意图（一）

图 1-14 中的运算接受两个输入（都可以是 `ndarray`），并生成一个输出 `ndarray`。

对涉及多个输入的大量运算而言，这确实做了很大的简化。但是我们也可以突出显示各个运算和输入，如图 1-15 和图 1-16 所示。

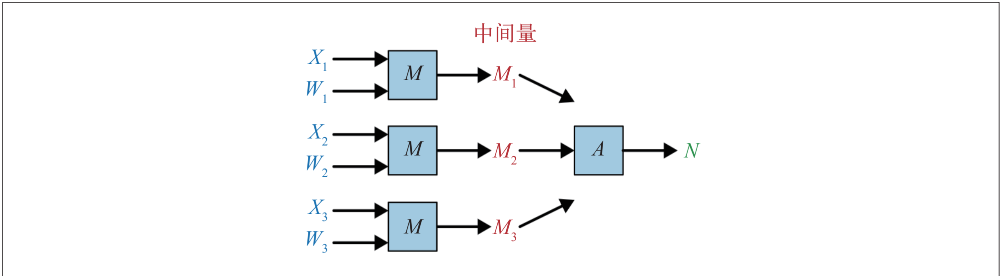


图 1-15：矩阵乘法示意图（二）

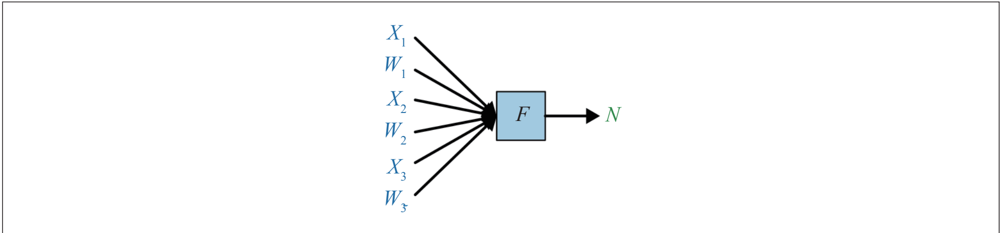


图 1-16：矩阵乘法示意图（三）

注意，矩阵乘法（向量点积）是表示许多独立运算的一种简洁方法。这种运算会让后向传递的导数计算起来非常简洁，下一节将介绍这一点。

代码

对矩阵乘法进行编码很简单：

```
def matmul_forward(X: ndarray,
                   W: ndarray) -> ndarray:
    '''
    计算矩阵乘法的前向传递结果。
    '''

    assert X.shape[1] == W.shape[0], \
    '''
    对于矩阵乘法，第一个数组中的列数应与第二个数组中的行数相匹配。而这里，
    第一个数组中的列数为{0}，第二个数组中的行数为{1}。'''
     .format(X.shape[1], W.shape[0])

    # 矩阵乘法
    N = np.dot(X, W)

    return N
```

这里有一个新的断言，它确保了矩阵乘法的有效性。（这是第一个运算，它不仅处理相同大小的 `ndarray`，还逐元素执行运算，而现在的输出与输入实际上并不匹配。因此，这个断言很重要。）

1.10 多向量输入函数的导数

对于仅以一个数字作为输入并生成一个输出的函数，例如 $f(x)=x^2$ 或 $f(x)=\text{sigmoid}(x)$，计算导数很简单，只需应用微积分中的规则即可。然而，对于向量函数，其导数就没有那么简单了：如果将点积写为 $v(\boldsymbol{X}, \boldsymbol{W})=N$ 这种形式，那么自然会产生一个问题：$\frac{\partial N}{\partial \boldsymbol{X}}$ 和 $\frac{\partial N}{\partial \boldsymbol{W}}$ 分别是什么？

数学

如何定义“矩阵的导数”？回顾一下，矩阵语法只是对一堆以特定形式排列的数字的简写，“矩阵的导数”实际上是指“矩阵中每个元素的导数”。由于 $\boldsymbol{X}$ 有一行，因此它可以这样定义：

$$\frac{\partial v}{\partial \boldsymbol{X}}=\begin{bmatrix}\frac{\partial v}{\partial x_1} & \frac{\partial v}{\partial x_2} & \frac{\partial v}{\partial x_3}\end{bmatrix}$$

然而，v 的输出只是一个数字：$N = x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$。可以看到，如果 x_1 改变了 ϵ 单位，那么 N 将改变 $w_1 \times \epsilon$ 单位。同理，其他的 x_i 元素也满足这种情况。因此可以得出下面的公式：

$$\frac{\partial v}{\partial x_1} = w_1$$

$$\frac{\partial v}{\partial x_2} = w_2$$

$$\frac{\partial v}{\partial x_3} = w_3$$

$$\frac{\partial v}{\partial \boldsymbol{X}} = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} = \boldsymbol{W}^{\mathrm{T}}$$

这个结果出乎意料地简练，掌握这一点极为关键，既可以理解深度学习的有效性，又可以知道如何清晰地实现深度学习。

以此类推，可以得到如下公式。

$$\frac{\partial v}{\partial \boldsymbol{W}} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \boldsymbol{X}^{\mathrm{T}}$$

示意图

从概念上讲，我们只想执行图 1-17 所示的操作。

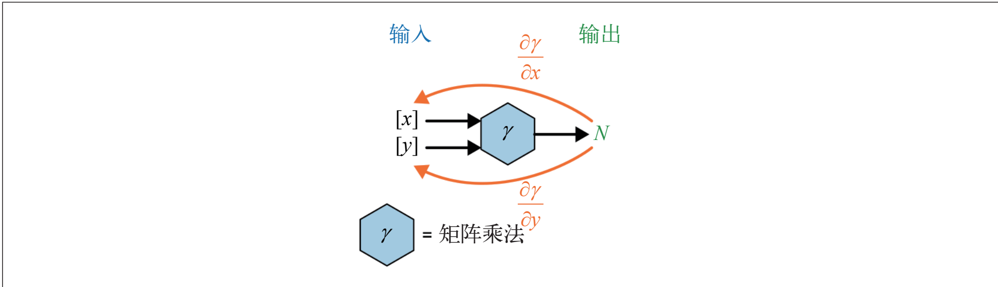


图 1-17：矩阵乘法的后向传递

如前面的例子所示，当只处理加法和乘法时，计算这些导数很容易。但是如何利用矩阵乘法实现类似的操作呢？图 1-17 中的示意图并不直观，要准确地定义它，必须求助于本节中的数学公式。

代码

从数学上推算答案应该是最困难的部分，对结果进行编码则比较简单：

```
def matmul_backward_first(X: ndarray,
                          W: ndarray) -> ndarray:
    '''
    计算矩阵乘法相对于第一个参数的后向传递结果。
    '''

    # 后向传递
    dNdX = np.transpose(W, (1, 0))

    return dNdX
```

这里计算的 dNdX 表示 $\boldsymbol{X}$ 的每个元素相对于输出 N 的和的偏导数。在本书中，这个量有一个特殊的名称，即 $\boldsymbol{X}$ 的**梯度**（gradient）。这个概念是指，对于 $\boldsymbol{X}$ 的单个元素（例如 x_3），dNdX 中的对应元素（具体来说是 dNdX[2]）是向量点积 N 的输出相对于 x_3 的偏导数。在本书中，梯度仅指偏导数的多维对应物。具体来说，它是函数输出相对于该函数输入的每个元素的偏导数数组。

1.11 向量函数及其导数：再进一步

当然，深度学习模型不止涉及一个运算，它们包括长链式运算，其中一些是 1.10 节介绍的向量函数，另一些则只是将函数逐元素地应用于它们接受的 ndarray（输入）中。现在来计算包含这两种函数的复合函数的导数。假设函数接受向量 $\boldsymbol{X}$ 和向量 $\boldsymbol{W}$，执行 1.10 节描述的点积（将其表示为 $v(\boldsymbol{X}, \boldsymbol{W})$），然后将向量输入到函数 σ 中。这里将用新的语言来表达同样的目标：计算这个新函数的输出相对于向量 $\boldsymbol{X}$ 和向量 $\boldsymbol{W}$ 的梯度。从第 2 章开始，本书将详细介绍它如何与神经网络相关联。现在只需大致了解这个概念，也就是可以为任意复杂度的计算图计算梯度。

数学

公式很简单，如下所示。

$$s = f(\boldsymbol{X}, \boldsymbol{W}) = \sigma(v(\boldsymbol{X}, \boldsymbol{W})) = \sigma(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3)$$

示意图

图 1-18 与图 1-17 类似，只不过在最后添加了函数 σ 。

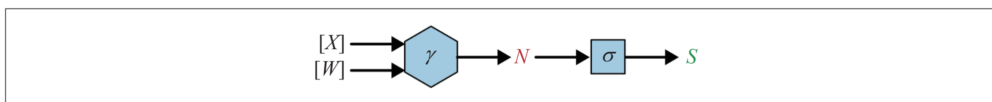


图 1-18：与图 1-17 类似，但在最后添加了另一个函数

代码

可以像下面这样编写本例中的函数。

```
def matrix_forward_extra(X: ndarray,
                         W: ndarray,
                         sigma: Array_Function) -> ndarray:
    '''
    计算涉及矩阵乘法的函数（一个额外的函数）的前向传递结果。
    '''
    assert X.shape[1] == W.shape[0]

    # 矩阵乘法
    N = np.dot(X, W)

    # 通过sigma传递矩阵乘法的输出
    S = sigma(N)

    return S
```

向量函数及其导数：后向传递

类似地，后向传递只是前述示例的直接扩展。

1. 数学

由于 $f(\boldsymbol{X}, \boldsymbol{W})$ 是嵌套函数，具体来说就是 $f(\boldsymbol{X}, \boldsymbol{W}) = \sigma(v(\boldsymbol{X}, \boldsymbol{W}))$，因此该函数在 $\boldsymbol{X}$ 处的导数可以这样表示：

$$\frac{\partial f}{\partial \boldsymbol{X}} = \frac{\partial \sigma}{\partial u}(v(\boldsymbol{X}, \boldsymbol{W})) \times \frac{\partial v}{\partial \boldsymbol{X}}(\boldsymbol{X}, \boldsymbol{W})$$

第一部分很简单：

$$\frac{\partial \sigma}{\partial u}(v(\boldsymbol{X}, \boldsymbol{W})) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3)$$

这是一个很好的定义，σ 是连续函数，可以在任意点求导。在这里，只在 $x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$ 处对其求值。

此外，我们在 1.10 节的示例中已经推断出 $\frac{\partial v}{\partial \boldsymbol{X}}(\boldsymbol{X}, \boldsymbol{W}) = \boldsymbol{W}^{\mathrm{T}}$。因此，可以这样表达：

$$\frac{\partial f}{\partial \boldsymbol{X}} = \frac{\partial \sigma}{\partial u}(v(\boldsymbol{X}, \boldsymbol{W})) \times \frac{\partial v}{\partial \boldsymbol{X}}(\boldsymbol{X}, \boldsymbol{W}) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3) \times \boldsymbol{W}^{\mathrm{T}}$$

与前面的示例一样，由于最终答案是数字 $\frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3)$ 乘以 $\boldsymbol{W}^{\mathrm{T}}$ 中的与 $\boldsymbol{X}$ 形状相同的向量，因此这个公式会得出一个与 $\boldsymbol{X}$ 形状相同的向量。

2. 示意图

图 1-19 所示的这个函数的后向传递示意图与前面的例子类似，甚至不需要在数学上做过多解释。矩阵乘法的结果包含所计算的 σ 函数的导数，只需要在这个导数的基础上再添加一个乘法。

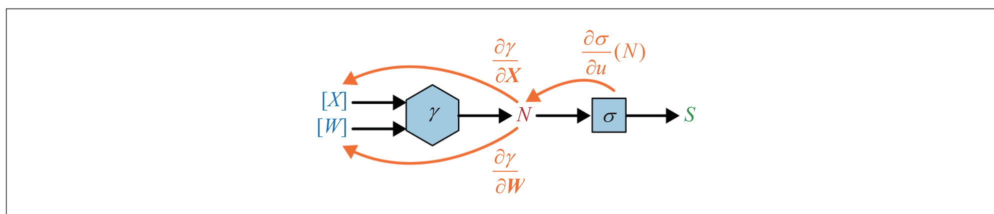


图 1-19：带有矩阵乘法的图：后向传递

3. 代码

对后向传递进行编码也很简单：

```
def matrix_function_backward_1(X: ndarray,
                               W: ndarray,
                               sigma: Array_Function) -> ndarray:
    '''
    计算矩阵函数相对于第一个元素的导数。
    '''
    assert X.shape[1] == W.shape[0]

    # 矩阵乘法
    N = np.dot(X, W)

    # 通过sigma传递矩阵乘法的输出
    S = sigma(N)

    # 后向计算
    dSdN = deriv(sigma, N)

    # dNdX
    dNdX = np.transpose(W, (1, 0))

    # 将它们相乘。因为这里的dNdX是1×1，所以顺序无关紧要
    return np.dot(dSdN, dNdX)
```

注意，这里显示的动态效果与 1.5 节中 3 个嵌套函数示例显示的动态效果相同：计算前向传递（这里指 N）上的量，然后在后向传递期间进行使用。

4. 这是对的吗？

如何判断正在计算的这些导数是否正确？测试起来很简单，就是稍微扰动输入并观察输出结果的变化。例如，在这种情况下，$\boldsymbol{X}$ 为：

```
print(X)
[[ 0.4723 0.6151 -1.7262]]
```

如果将 x_3 增加 0.01，即从 `-1.7262` 增加到 `-1.7162`，那么应该可以看到由输出梯度相对于 $x_3 \times 0.01$ 的前向函数生成的值有所增加，如图 1-20 所示。

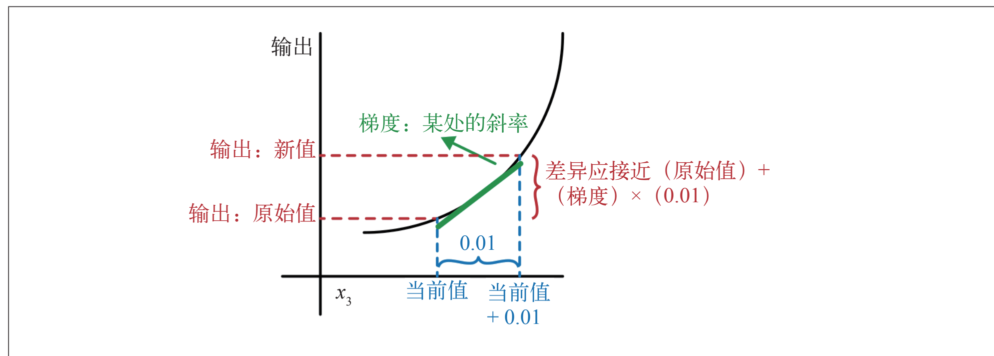


图 1-20：梯度检查示意图

利用 `matrix_function_backward_1` 函数，可以看到梯度是 `-0.1121`：

```
print(matrix_function_backward_1(X, W, sigmoid))
[[ 0.0852 -0.0557 -0.1121]]
```

可以看到，在将 x_3 递增 0.01 之后，函数的输出相应减少了约 `0.01 × -0.1121 = -0.001121`，这可以帮助测试该梯度是否正确。如果减幅（或增幅）大于或小于此量，那么关于链式法则的计算就是错误的。然而，当执行计算时，少量增加 x_3 确实会使函数的输出值减小 `0.01 × -0.1121`，这意味着计算的导数是正确的！

1.12 节介绍的示例涉及前面介绍的所有运算，并且可以直接用于第 2 章将构建的模型。

1.12　包含两个二维矩阵输入的计算图

在深度学习和更通用的机器学习中，需要处理输入为两个二维数组的运算，其中一个数组表示一批数据 $\boldsymbol{X}$，另一个表示权重 $\boldsymbol{W}$。这对建模上下文很有帮助，第 2 章将对此展开介绍。本章仅关注此运算背后的原理和数学意义，具体来说，就是通过一个简单的示例详细说明，我们不再以一维向量的点积为例，而是介绍二维矩阵的乘法。即便如此，本章介绍的推算过程仍然具有数学意义，并且实际上非常容易编码。

和以前一样，从数学上看，得出这些结果并不困难，但过程看起来有点复杂。不管怎样，结果还是相当清晰的。当然，我们会对其按步骤进行分解，并将其与代码和示意图联系起来。

数学

假设 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 如下所示：

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

这可能对应一个数据集，其中每个观测值都具有 3 个特征，3 行可能对应要对其进行预测的 3 个观测值。

现在将为这些矩阵定义以下简单的运算。

1. 将这些矩阵相乘。和以前一样，将把执行此运算的函数表示为 $v(\boldsymbol{X}, \boldsymbol{W})$，将输出表示为 N。因此，可以这样表示：$N = v(\boldsymbol{X}, \boldsymbol{W})$。
2. 将结果 N 传递给可微函数 σ，并定义 $S = \sigma(N)$。

和以前一样，现在的问题是：输出 S 相对于 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 的梯度是多少？可以简单地再次使用链式法则吗？为什么？

注意，本例与之前的示例有所不同：S 不是数字，而是矩阵。那么，一个矩阵相对于另一矩阵的梯度意味着什么呢？

这就引出了一个微妙但十分重要的概念：可以在目标多维数组上执行任何一系列运算，但是要对某些输出定义好梯度，这需要对序列中的最后一个数组**求和**（或以其他方式聚合成单个数字），这样“$\boldsymbol{X}$ 中每个元素的变化会在多大程度上影响输出”这一问题才有意义。

因此，在最后添加第 3 个函数 Λ，该函数获取 S 中的元素并将其求和。

通过数学把它具体化。首先，把 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 相乘：

$$
\boldsymbol{X}\times\boldsymbol{W}=\begin{bmatrix}x_{11}\times w_{11}+x_{12}\times w_{21}+x_{13}\times w_{31} & x_{11}\times w_{12}+x_{12}\times w_{22}+x_{13}\times w_{32}\\ x_{21}\times w_{11}+x_{22}\times w_{21}+x_{23}\times w_{31} & x_{21}\times w_{12}+x_{22}\times w_{22}+x_{23}\times w_{32}\\ x_{31}\times w_{11}+x_{32}\times w_{21}+x_{33}\times w_{31} & x_{31}\times w_{12}+x_{32}\times w_{22}+x_{33}\times w_{32}\end{bmatrix}
=\begin{bmatrix}XW_{11} & XW_{12}\\ XW_{21} & XW_{22}\\ XW_{31} & XW_{32}\end{bmatrix}
$$

为了便于书写结果矩阵，这里将第 i 行的第 j 列表示为 XW_{ij} 。

接下来，将该结果输入到 σ 中，这意味着将 σ 应用于 $\boldsymbol{X}\times\boldsymbol{W}$ 矩阵中的每个元素：

$$
\sigma(\boldsymbol{X}\times\boldsymbol{W})=\begin{bmatrix}\sigma(x_{11}\times w_{11}+x_{12}\times w_{21}+x_{13}\times w_{31}) & \sigma(x_{11}\times w_{12}+x_{12}\times w_{22}+x_{13}\times w_{32})\\ \sigma(x_{21}\times w_{11}+x_{22}\times w_{21}+x_{23}\times w_{31}) & \sigma(x_{21}\times w_{12}+x_{22}\times w_{22}+x_{23}\times w_{32})\\ \sigma(x_{31}\times w_{11}+x_{32}\times w_{21}+x_{33}\times w_{31}) & \sigma(x_{31}\times w_{12}+x_{32}\times w_{22}+x_{33}\times w_{32})\end{bmatrix}
=\begin{bmatrix}\sigma(XW_{11}) & \sigma(XW_{12})\\ \sigma(XW_{21}) & \sigma(XW_{22})\\ \sigma(XW_{31}) & \sigma(XW_{32})\end{bmatrix}
$$

最后，对这些元素求和：

$$
L=\Lambda(\sigma(\boldsymbol{X}\times\boldsymbol{W}))=\Lambda\left(\begin{bmatrix}\sigma(XW_{11}) & \sigma(XW_{12})\\ \sigma(XW_{21}) & \sigma(XW_{22})\\ \sigma(XW_{31}) & \sigma(XW_{32})\end{bmatrix}\right)=\sigma(XW_{11})+\sigma(XW_{12})+\sigma(XW_{21})+
\sigma(XW_{22})+\sigma(XW_{31})+\sigma(XW_{32})
$$

现在回到了纯微积分的场景中：存在一个数字 L，想计算出 L 相对于 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 的梯度，也就是明确这些输入矩阵中**每个元素**（x_{11}、w_{21} 等）的变化对 L 的影响。可以这样写：

$$
\frac{\partial\Lambda}{\partial u}(\boldsymbol{X})=\begin{bmatrix}\frac{\partial\Lambda}{\partial u}(x_{11}) & \frac{\partial\Lambda}{\partial u}(x_{12}) & \frac{\partial\Lambda}{\partial u}(x_{13})\\ \frac{\partial\Lambda}{\partial u}(x_{21}) & \frac{\partial\Lambda}{\partial u}(x_{22}) & \frac{\partial\Lambda}{\partial u}(x_{23})\\ \frac{\partial\Lambda}{\partial u}(x_{31}) & \frac{\partial\Lambda}{\partial u}(x_{32}) & \frac{\partial\Lambda}{\partial u}(x_{33})\end{bmatrix}
$$

至此，我们已经从数学上理解了所面临的问题，接下来讨论示意图层面和代码层面。

示意图

从概念上讲，与前面介绍的多输入函数的计算图相比，包含两个二维矩阵输入的运算所做的工作其实是类似的，如图 1-21 所示。

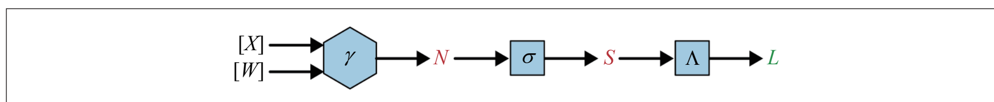


图 1-21：具有复杂前向传递函数的计算图

这里只是像以前一样向前发送输入。有一点需要明确：即使在这个更复杂的场景中，也应该能够使用链式法则计算所需的梯度。

代码

对于输入为两个二维数组的运算，可以像下面这样编码。

```
def matrix_function_forward_sum(X: ndarray,
                                W: ndarray,
                                sigma: Array_Function) -> float:
    '''
    输入ndarray（X、W）以及函数sigma，计算该函数的前向传递结果。
    '''
    assert X.shape[1] == W.shape[0]

    # 矩阵乘法
    N = np.dot(X, W)

    # 通过sigma传递矩阵乘法的输出
    S = sigma(N)

    # 将所有元素相加
    L = np.sum(S)

    return L
```

1.13　有趣的部分：后向传递

现在，要对函数“执行后向传递”，这样一来，即使涉及矩阵乘法，也可以最终计算出 N 相对于输入 ndarray 的每个元素的梯度[5]。在学完本章之后，就能轻松地在第 2 章中开始训练真正的机器学习模型。下面先从概念上明确要学习的内容。

数学

注意，可以直接计算。值 L 实际上是从 x_{11}、x_{12} 直到 x_{33} 的一个函数。

但是，这似乎很复杂。链式法则的全部要点就是将复杂函数的导数分解成简单的部分，对每个部分执行计算，然后把结果相乘。如此一来，对这些操作进行编码就变得很容易：只需要逐步进行前向传递，保存传递过程中的结果，然后使用这些结果来计算后向传递所需的所有导数。

注 5：接下来重点计算 N 相对于 X 的梯度，但 W 的梯度可以通过类似的方法来计算。

下面展示这种方法**仅适用于涉及矩阵的情况**。开始深入讨论吧。

可以将 L 写成 $\Lambda(\sigma(v(\boldsymbol{X}, \boldsymbol{W})))$ 。如果这是一个常规函数，就可以这样编写链式法则：

$$\frac{\partial\Lambda}{\partial\boldsymbol{X}}(\boldsymbol{X})=\frac{\partial v}{\partial\boldsymbol{X}}(\boldsymbol{X}, \boldsymbol{W})\times\frac{\partial\sigma}{\partial u}(N)\times\frac{\partial\Lambda}{\partial u}(S)$$

然后依次计算 3 个偏导数。前面在计算包含 3 个嵌套函数的复合函数的导数时，我们使用链式法则分别对每个函数进行了求导，这里要执行同样的操作。图 1-22（参见下一页）表明，该方法同样适用于这种函数。

由于一阶导数最直接，因此这里从一阶导数开始计算，主要是确定当 S 中每个元素值增加时，Λ 的输出 L 的增长情况。由于 L 是 S 中所有元素的总和，因此这个导数很简单：

$$\frac{\partial\Lambda}{\partial u}(S)=\begin{bmatrix}1 & 1\\1 & 1\\1 & 1\end{bmatrix}$$

只要 S 中的任何元素有所增加，比如增加 0.46 个单位，Λ 就会增加 0.46 单位。

接下来得到 $\frac{\partial\sigma}{\partial u}(N)$ 。这只是对 N 中元素进行求值的任一函数 σ 的导数。在前面使用的 XW 语法中，这同样很容易计算：

$$\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{11}) & \frac{\partial\sigma}{\partial u}(XW_{12})\\\frac{\partial\sigma}{\partial u}(XW_{21}) & \frac{\partial\sigma}{\partial u}(XW_{22})\\\frac{\partial\sigma}{\partial u}(XW_{31}) & \frac{\partial\sigma}{\partial u}(XW_{32})\end{bmatrix}$$

注意，我们现在可以肯定地说，能够将这两个导数按元素逐个相乘并计算 $\frac{\partial\Lambda}{\partial u}(N)$ ：

$$\frac{\partial\Lambda}{\partial u}(N)=\frac{\partial\Lambda}{\partial u}(S)\times\frac{\partial\sigma}{\partial u}(N)=\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{11}) & \frac{\partial\sigma}{\partial u}(XW_{12})\\\frac{\partial\sigma}{\partial u}(XW_{21}) & \frac{\partial\sigma}{\partial u}(XW_{22})\\\frac{\partial\sigma}{\partial u}(XW_{31}) & \frac{\partial\sigma}{\partial u}(XW_{32})\end{bmatrix}\times\begin{bmatrix}1 & 1\\1 & 1\\1 & 1\end{bmatrix}=\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{11}) & \frac{\partial\sigma}{\partial u}(XW_{12})\\\frac{\partial\sigma}{\partial u}(XW_{21}) & \frac{\partial\sigma}{\partial u}(XW_{22})\\\frac{\partial\sigma}{\partial u}(XW_{31}) & \frac{\partial\sigma}{\partial u}(XW_{32})\end{bmatrix}$$

然而，现在陷入了困境。基于图 1-22 并应用链式法则，接下来要做的是获取 $\frac{\partial v}{\partial u}(\boldsymbol{X})$ 。但是，回想一下，v 的输出 N 只是 $\boldsymbol{X}$ 与 $\boldsymbol{W}$ 矩阵相乘的结果。因此，这里要知道 N（3×2 矩阵）中每个元素随着 $\boldsymbol{X}$ 中每个元素（3×3 矩阵）的增加而增加的量。如果上面的表述难以理解，只要明确一点就可以了，那就是现在根本不清楚如何定义它，或者无法确定这样做真的有效。

为什么会出现这个问题呢？以前，我们很幸运，由于 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 在形状上可以相互转换，因此可以证明 $\frac{\partial v}{\partial u}(\boldsymbol{X})=\boldsymbol{W}^{\mathrm{T}}$ 和 $\frac{\partial v}{\partial u}(\boldsymbol{W})=\boldsymbol{X}^{\mathrm{T}}$ 。现在可以得出类似的结论吗？

“？”的值

更具体地说，现在需要弄清楚以下公式中的“？”到底是什么。

$$\frac{\partial \Lambda}{\partial u}(\boldsymbol{X})=\frac{\partial \Lambda}{\partial u}(\sigma(N))\times ?=\begin{bmatrix}\frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12})\\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22})\\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32})\end{bmatrix}\times ?$$

答案

事实证明，根据乘法的计算方式，“？”处的内容就是 $\boldsymbol{W}^{\mathrm{T}}$，这和刚才看到的向量点积的简单示例是一样的。有一种方法可以验证这一点，那就是直接针对 $\boldsymbol{X}$ 中的每个元素计算 L 的偏导数。这样一来[6]，得到的矩阵确实显著地分解成：

$$\frac{\partial \Lambda}{\partial u}(\boldsymbol{X})=\frac{\partial \Lambda}{\partial u}(S)\times\frac{\partial \sigma}{\partial u}(N)\times\boldsymbol{W}^{\mathrm{T}}$$

其中第一个乘法是逐个元素执行的，第二个则是矩阵乘法。

这意味着，**即使计算图中的运算涉及将矩阵与多行和多列相乘，并且即使这些运算的输出形状与输入的形状不同，仍然可以将这些运算包含在计算图中，并且使用“链式法则”逻辑对它们进行反向传播**。这个结果非常重要，如果没有这个结果，那么训练深度学习模型将变得更加烦琐，后文会进一步介绍这一点。

示意图

本例的示意图与 1.12 节中的类似，如图 1-22 所示。

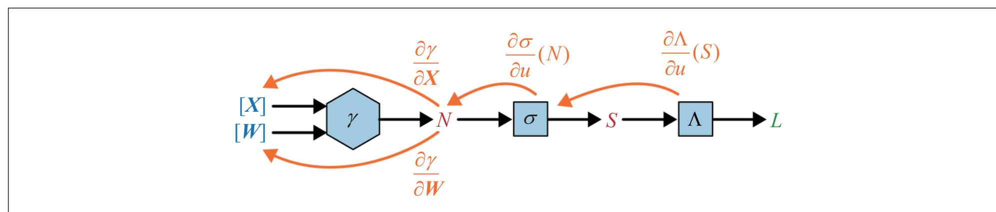


图 1-22：复杂函数中的后向传递

注 6：更多介绍参见附录的“矩阵链式法则”。

只需要计算每个组成函数的偏导数，在其输入处进行求值，并将结果相乘，就可以得到最终的导数。要依次计算这些偏导数，唯一的方法就是从上述数学层面计算。

代码

现在通过代码封装前面推导出的内容，此过程有助于加深对内容的理解：

```
def matrix_function_backward_sum_1(X: ndarray,
                                   W: ndarray,
                                   sigma: Array_Function) -> ndarray:
    '''
    计算矩阵函数相对于第一个矩阵输入的和的导数。
    '''
    assert X.shape[1] == W.shape[0]

    # 矩阵乘法
    N = np.dot(X, W)

    # 通过sigma传递矩阵乘法的输出
    S = sigma(N)

    # 将所有元素相加
    L = np.sum(S)

    # 注意，这里按数量指代导数，这点与数学不同，在数学中使用的是它们的函数名

    # dLdS——都是1
    dLdS = np.ones_like(S)

    # dSdN
    dSdN = deriv(sigma, N)

    # dLdN
    dLdN = dLdS * dSdN

    # dNdX
    dNdX = np.transpose(W, (1, 0))

    # dLdX
    dLdX = np.dot(dSdN, dNdX)

    return dLdX
```

现在，确认一切正常：

```
np.random.seed(190204)
X = np.random.randn(3, 3)
W = np.random.randn(3, 2)

print("X:")
print(X)

print("L:")
```

```
print(round(matrix_function_forward_sum(X, W, sigmoid), 4))
print()
print("dLdX:")
print(matrix_function_backward_sum_1(X, W , sigmoid))

X:
[[-1.5775 -0.6664  0.6391]
 [-0.5615  0.7373 -1.4231]
 [-1.4435 -0.3913  0.1539]]
L:
2.3755

dLdX:
[[ 0.2489 -0.3748  0.0112]
 [ 0.126  -0.2781 -0.1395]
 [ 0.2299 -0.3662 -0.0225]]
```

和前面的示例一样，由于 dLdX 表示 $\boldsymbol{X}$ 相对于 L 的梯度，因此这意味着，左上角的元素表示 $\frac{\partial \Lambda}{\partial x_{11}}(\boldsymbol{X}, \boldsymbol{W}) = 0.2489$ 。

如果这个示例的矩阵数学是正确的，则将 x_{11} 增加 0.001 会导致 L 增加 0.01×0.2489 。事实上，代码的运行情况是这样的：

```
X1 = X.copy()
X1[0, 0] += 0.001

print(round(
        (matrix_function_forward_sum(X1, W, sigmoid) - \
         matrix_function_forward_sum(X, W, sigmoid)) / 0.001, 4))
0.2489
```

看起来梯度的计算是正确的！

直观地描述梯度

回到前面提到的内容，将元素 x_{11} 传递给具有多重运算的函数，包括矩阵乘法、sigmoid 函数、求和运算。其中的矩阵乘法实际上是由矩阵 $\boldsymbol{X}$ 中的 9 个输入与矩阵 $\boldsymbol{W}$ 中的 6 个输入相结合，从而创建出的 6 个输出的简写。然而，也可以将其视为单独的函数 WNSL，如图 1-23 所示。

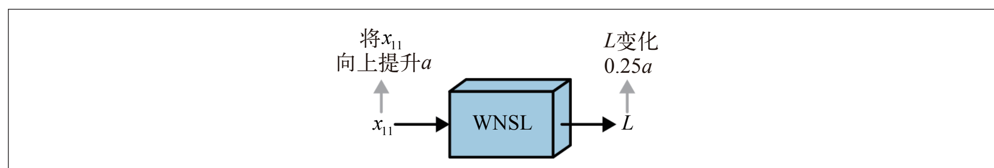


图 1-23：用单个函数 WNSL 来描述嵌套函数

由于每个函数都是可微的，因此整个函数就是一个以 x_{11} 为输入的可微函数。另外，梯度就是 $\frac{dL}{dx_{11}}$ 。为了使其可视化，可以简单地绘制 L 随着 x_{11} 的变化而变化的情况。

x_{11}的初始值是 -1.5775：

```
print("X:")
print(X)

X:
[[-1.5775 -0.6664  0.6391]
 [-0.5615  0.7373 -1.4231]
 [-1.4435 -0.3913  0.1539]]
```

对于将 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 输入到前面定义的计算图中，或者说将 $\boldsymbol{X}$ 和 $\boldsymbol{W}$ 输入到前面代码调用的函数中，如果绘制整个过程中所得到的 L 值的图像，并且除了 x_{11}（X[0, 0]）外不做任何变动，可以得到图 1-24 所示的结果[7]。

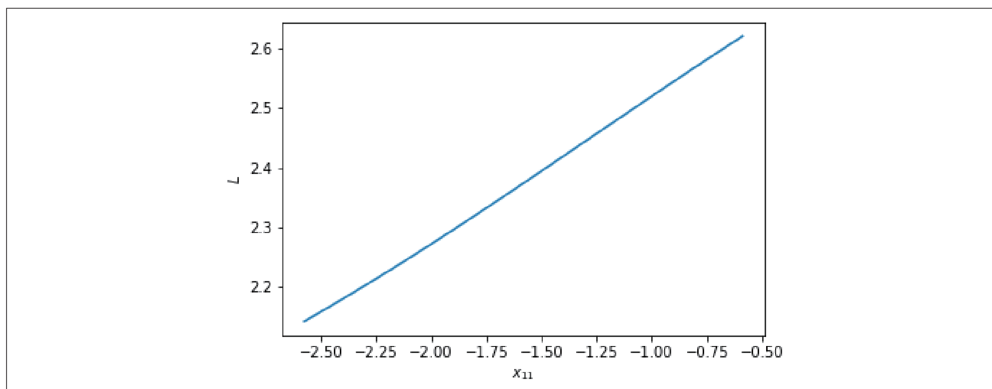


图 1-24：在保持 X 和 W 的值为常数的情况下，L 与 x_{11} 的对应关系

确实，在只变动 x_{11} 的情况下，这种关系看起来很明显。可以看到，此函数在纵轴上大约增加了 0.5（从刚好超过 2.1 到刚好超过 2.6），并且在横轴上大约增加了 2。因此，斜率大约为 $\frac{0.5}{2}=0.25$，这正是刚刚计算的结果！

复杂的矩阵数学事实上正确地计算了 L 相对于 $\boldsymbol{X}$ 中所有元素的偏导数。此外，可以用类似的方法计算 L 相对于 $\boldsymbol{W}$ 的梯度。



L 相对于 $\boldsymbol{W}$ 的梯度的表达式为 $\boldsymbol{X}^{\mathrm{T}}$。但是，$\boldsymbol{X}^{\mathrm{T}}$ 表达式中的因子是从 L 的导数中导出的，考虑到它们的顺序，$\boldsymbol{X}^{\mathrm{T}}$ 将位于 L 相对于 $\boldsymbol{W}$ 的梯度的表达式的左侧：

$$\frac{\partial \Lambda}{\partial u}(\boldsymbol{W})=\boldsymbol{X}^{\mathrm{T}}\times\frac{\partial \Lambda}{\partial u}(S)\times\frac{\partial \sigma}{\partial u}(N)$$

因此，尽管代码中出现了 `dNdX = np.transpose(X, (1, 0))`，但下一步将是：

```
dLdW = np.dot(dNdX, dSdN)
```

而不是之前的 `dLdX = np.dot(dSdN, dNdX)`。

注 7：这里展示的只是 `matrix_function_backward_sum` 函数的子集。完整函数可以从图灵社区下载：ituring.cn/book/2759。——编者注

1.14　小结

学完本章之后，你应该有信心理解复杂的嵌套数学函数，并通过将它们概念化为一系列盒子来解释它们的工作原理，每个盒子代表一个由线连接的单一组成函数。尤其需要注意的是，即使存在涉及二维 `ndarray` 的矩阵乘法，也可以编写代码来计算这些函数的输出相对于任何输入的导数，理解正确计算这些导数背后的数学原理。掌握这些基本概念，有助于接下来构建和训练神经网络。加油！

第 2 章

基本原理

第 1 章介绍了深度学习主要的构成要素，包括嵌套、连续和可微的函数，展示了如何将这些函数表示为计算图（图中的每个节点代表一个简单函数），特别论证了在计算嵌套函数的输出相对于其输入的导数时，计算图很容易展示计算过程，即只需提取所有组成函数的导数，在这些函数接收输入时计算这些导数，并将所有结果相乘[1]。论证过程涉及一些示例，这些示例中的函数以 NumPy 库的 `ndarray` 作为输入，并将生成的 `ndarray` 作为输出。当然，还有一些其他示例，它们说明即使函数将多个 `ndarray` 作为输入并通过**矩阵乘法**将它们组合在一起，这种计算导数的方法仍然有效。矩阵乘法与其他运算不同，它可以改变其输入的形状。具体地说，如果此运算的一个输入是 $\boldsymbol{X}$（$\boldsymbol{B} \times \boldsymbol{N}$ 的 `ndarray`），另一个输入是 $\boldsymbol{W}$（$\boldsymbol{N} \times \boldsymbol{M}$ 的 `ndarray`），其输出 $\boldsymbol{P}$ 就是一个 $\boldsymbol{B} \times \boldsymbol{M}$ 的 `ndarray`[2]。虽然该运算的导数尚不明确，但可以看到，当矩阵乘法 $v(\boldsymbol{X}, \boldsymbol{W})$ 作为一种“组合运算”包含在嵌套函数中时，仍然可以使用一种简单的表达式**代替**导数来计算其输入的导数：确切地讲，就是 $\boldsymbol{X}^{\mathrm{T}}$ 可以代替 $\frac{\partial v}{\partial u}(\boldsymbol{W})$，$\boldsymbol{W}^{\mathrm{T}}$ 则可以代替 $\frac{\partial v}{\partial u}(\boldsymbol{X})$。

本章将开始把这些概念转换为真实的应用程序，具体涉及以下几项操作。

1. 根据第 1 章介绍的构成要素解释线性回归。
2. 证明在第 1 章中所做的关于导数的计算能够训练这种线性回归模型。
3. 运用同样的构成要素将此模型扩展为单层神经网络。

当完成以上操作后，就能轻松地在第 3 章中运用相同的构成要素来构建深度学习模型。

注 1：基于链式法则，通过这种方式生成的嵌套函数的导数是正确的。

注 2：这里的 $\boldsymbol{B}$、$\boldsymbol{N}$、$\boldsymbol{M}$ 分别是不同的自然数。

不过，在深入探讨这些内容之前，先简要描述一下监督学习。**监督学习**（supervised learning）是机器学习的子集，下面将在研究如何使用神经网络解决问题时对其着重研究。

2.1 监督学习概述

概括地讲，可以将机器学习描述为构建一些算法，这些算法可以发掘或“学习”数据中的**关系**（relationship）。监督学习是机器学习的子集，专用于发现**已测量的数据属性**之间的关系[3]。

本章将解决实际可能遇到的一个典型的监督学习问题：找到房屋的属性与价值之间的关系。显然，像房间数量、占地面积、是否靠近学校以及对居住或拥有房屋的渴望程度等，这些属性之间存在一定的关系。总体来说，鉴于已经测量了这些属性，监督学习的目的就是发掘这些属性之间的关系。

所谓“测量”，指的是每个属性均已被精确地定义并表示为一个数字。房屋的许多属性，比如房间数量、占地面积等，自然适合用数字进行表示，但是对于其他不同类型的属性，比如 TripAdvisor[4] 网站上对某地周边环境的整体评价，就不容易用数字表示了。而且，如果以一种合理的方式将这些不太结构化的数据转换成数字，那么可能会影响发掘关系的进程。另外，针对任何类似房屋价值这种非具象的概念，如果必须选择一个数字来描述它，可以选择使用房屋的价格[5]。

一旦把“属性”转换成数字，就必须决定使用哪种结构来表示这些数字。在机器学习中，有一个通用的方法能简化计算过程，那就是将单个观测值（例如一所房屋）的每组数字表示成**一行数据**，然后将这些行堆叠在一起形成数据的“批次”，这些数据将以二维 `ndarray` 的形式输入到模型中。模型将输出 `ndarray`，也就是返回预测结果（每一行代表一个预测结果），这些预测结果也彼此叠加，批次中的每个观测值对应一个预测结果。

现在做一些定义：`ndarray` 中每一行的长度是数据的**特征数**。一般来说，单个属性可以映射到多个特征，典型的示例是一个属性将数据描述为某几种类别中的一种[6]，例如红砖房、棕砖房或石板房。在这个特定的示例中，可以用 3 个特征来描述这个属性。将主观层面的（非正式的）观测结果的属性映射到特征的过程称为**特征工程**（feature engineering），但本书不会过多讨论这个过程。本章将解决一个问题，其中每个观测结果都有 13 个属性，每

注 3：无监督学习（unsupervised learning）是另一种机器学习，可以认为它是在已测量的事物和尚未测量的事物之间寻找关系。

注 4：TripAdvisor 是一个旅行平台，中文名为“猫途鹰”。——译者注

注 5：即使是在实际问题中，如何选择价格也不容易：选择最近一次出售房屋的价格吗？如果是一套很久没有出售过的房子，该如何选择呢？在本书的示例中，数据可以明确用数字表示或已经提前确定了，但是在许多实际问题中，正确地做到这一点并不简单。

注 6：大多数人可能知道，这些叫作“分类特征”。

个属性都仅用一个数值特征来表示。

前面说过，监督学习的最终目标是发掘数据属性之间的关系。在实践中，为了做到这一点，需要选择一个希望基于其他属性而进行预测的属性，这个属性称作**目标**（target）。任意属性都可以用作目标，具体取决于要解决的问题。如果目标只描述房屋价格和房间数量之间的关系，则可以训练一个模型，让其以房屋价格为目标，然后把房间数量作为一个特征，反之亦然。无论哪种方式，生成的模型实际上都包含对这两个属性之间关系的描述，例如可以说，房间的数量越多，房屋的价格就越高。但是，如果目标是**预测**房屋的价格，但又**没有可用的价格信息**，则必须选择价格作为目标，这样经过训练最终就可以将其他信息输入到模型中。

图 2-1 展示了描述监督学习的层次结构，从发现数据关系的最高层次描述，到最低层次的通过训练模型来量化这些关系，旨在揭示特征和目标之间的数值表示。

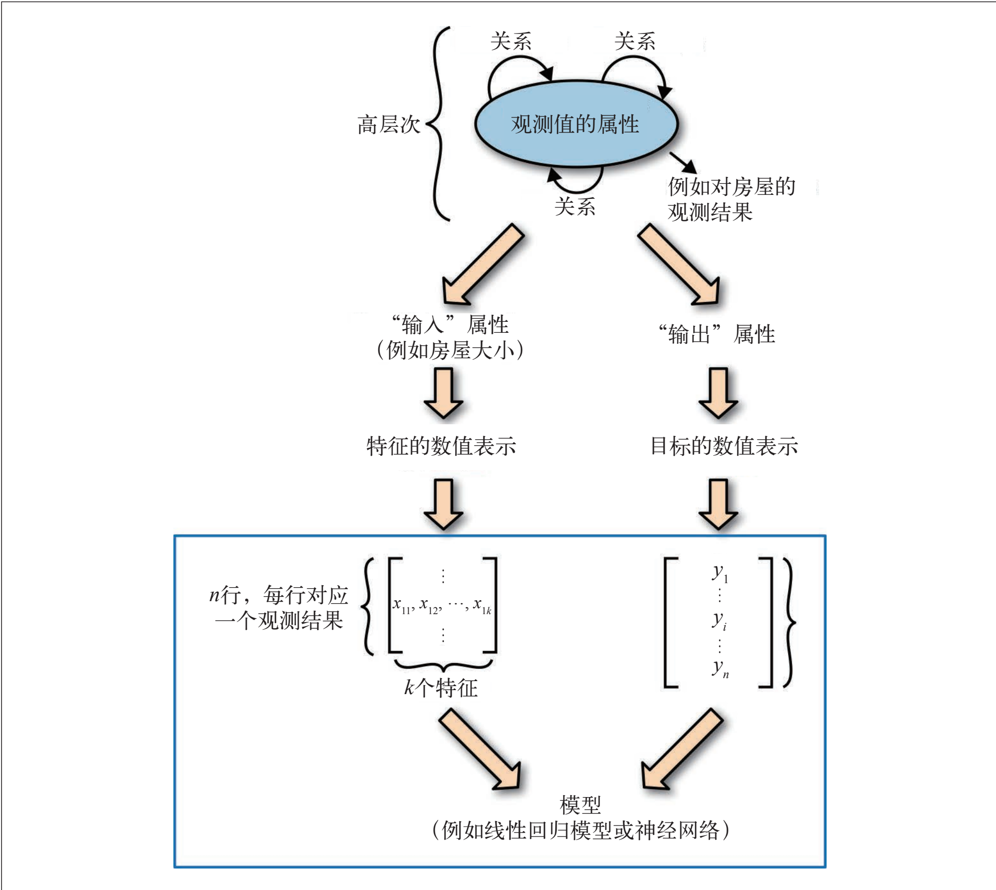


图 2-1：监督学习概述

如图 2-1 所示，本书侧重于介绍底部突出显示的层次。但是，在很多问题中，保证最上面的部分准确无误，涉及收集正确的数据、定义试图要解决的问题以及开展特征工程，这比实际建模要困难得多。不过，由于本书侧重建模，尤其侧重于介绍深度学习模型的工作原理，因此接下来继续讨论该主题。

2.2 监督学习模型

我们已经大致了解了监督学习模型的目标，正如本章已经提到的，这些模型只是嵌套的数学函数，第 1 章讨论了如何用数学、示意图和代码来表示这些函数。因此，本章可以更精确地用数学和代码（稍后将展示大量示意图）来解释监督学习的目标：**找到以 `ndarray` 为输入和输出的数学函数，该函数可以将观测值的属性映射到目标，即给定包含所创建特征的输入 `ndarray`，生成输出 `ndarray`，其值“贴近”包含目标的 `ndarray`**。

具体而言，就是用一个矩阵 $\boldsymbol{X}$ 表示数据，该矩阵有 n 行，每行代表一个具有 k 个特征的观测值，所有这些特征都是数字。每一行的观测值将是以 $x_i = [x_{i1}\ x_{i2}\ x_{i3} \cdots x_{ik}]$ 表示的向量，这些观测值将相互堆叠在一起形成一个批次。例如，下面是一个大小为 3 的批次：

$$\boldsymbol{X}_{\text{batch}} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \cdots & x_{1k} \\ x_{21} & x_{22} & x_{23} \cdots & x_{2k} \\ x_{31} & x_{32} & x_{33} \cdots & x_{3k} \end{bmatrix}$$

每一批观测值都会有相应的一批目标，其中的每个元素都是对应观测值的目标数字。可以将它们表示为一维向量：

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

就这些数组而言，监督学习的目标是使用第 1 章介绍的工具来构建一个函数，该函数可以将基于 $\boldsymbol{X}_{\text{batch}}$ 结构的观测值作为输入批次，然后生成值为 P_i 的向量，这个过程称为**预测**。至少对于特定数据集 $\boldsymbol{X}$ 中的数据而言，基于某种合理的**贴近度**（closeness）度量方法，这些预测在一定程度上“贴近”目标值 y_i 。

最后将所有这些具体化，并开始为真实的数据集构建第一个模型。接下来将从一个简单的线性回归模型入手，并展示如何用第 1 章介绍的构成要素来表达它。

2.3 线性回归

线性回归（linear regression）通常表示为如下形式：

$$y_i = \beta_0 + \beta_1 \times x_1 + \cdots + \beta_n \times x_k + \epsilon$$

该表示法从数学上描述了这样一个观点：每个目标的数值是 $\boldsymbol{X}$ 的 k 个特征的线性组合，再加上 β_0 项以调整预测的基线值，即当所有特征的值为 0 时所做的预测。

当然，这并不能帮助你深入理解如何编写代码以“训练”这样一个模型。要实现这一点，必须将这个模型转换成第 1 章提到的函数语言，最好从一个示意图开始。

2.3.1 线性回归：示意图

如何将线性回归表示为计算图？可以将其分解为多个独立元素，每个 x_i 与另一个元素 相乘，然后将结果相加在一起，如图 2-2 所示。

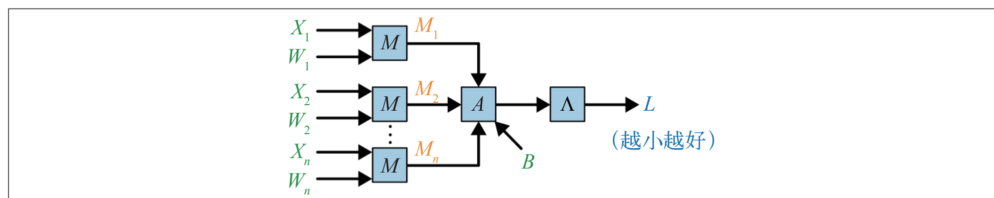


图 2-2：从乘法和加法的层面理解线性回归运算

正如在第 1 章中看到的，如果能把这些运算表示成矩阵乘法，就能更简洁地编写函数，同时还能正确地计算输出相对于输入的导数，这样便可以训练模型了。

应该怎么做？首先，处理一个较为简单的场景，在这个场景中没有截距项，如前面显示的 β_0。注意，可以将线性回归模型的输出表示为每个观测向量 $x_i = [x_1\, x_2\, x_3 \cdots x_k]$ 与另一个参数向量的**点积**，我们称这个参数向量为 $\boldsymbol{W}$：

$$\boldsymbol{W} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix}$$

如此一来，预测就很简单：

$$P_i = x_i \times \boldsymbol{W} = w_1 \times x_{i1} + w_2 \times x_{i2} + \cdots + w_k \times x_{ik}$$

因此，可以使用一个点积运算来表示线性回归的“生成预测”。

此外，当要使用一批观测值进行线性回归预测时，可以使用另一个运算：矩阵乘法。如果有一个像下面这样的大小为 3 的批次：

$$\boldsymbol{X}_{\text{batch}} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \cdots & x_{1k} \\ x_{21} & x_{22} & x_{23} \cdots & x_{2k} \\ x_{31} & x_{32} & x_{33} \cdots & x_{3k} \end{bmatrix}$$

执行此批次 $\boldsymbol{X}_{\text{batch}}$ 与 $\boldsymbol{W}$ 的矩阵乘法，即可根据需要给出该批次的预测向量：

$$\boldsymbol{P}_{\text{batch}} = \boldsymbol{X}_{\text{batch}} \times \boldsymbol{W} = \begin{bmatrix} x_{11} & x_{12} & x_{13} \cdots & x_{1k} \\ x_{21} & x_{22} & x_{23} \cdots & x_{2k} \\ x_{31} & x_{32} & x_{33} \cdots & x_{3k} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix}$$

$$= \begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13} \times w_3 + \cdots + x_{1k} \times w_k \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23} \times w_3 + \cdots + x_{2k} \times w_k \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33} \times w_3 + \cdots + x_{3k} \times w_k \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

因此，可以使用矩阵乘法完成对线性回归中的一批观测值的预测。接下来将展示如何利用这一事实以及第 1 章中关于导数的计算来训练该模型。

“训练”模型

“训练”一个模型意味着什么呢？概括地讲，模型接受数据，并以某种方式将其与**参数**（parameter）进行组合，以生成预测结果[7]。例如，前面展示的线性回归模型接受数据 $\boldsymbol{X}$ 和参数 $\boldsymbol{W}$，并使用矩阵乘法产生预测 $\boldsymbol{P}_{\text{batch}}$：

$$\boldsymbol{P}_{\text{batch}} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

但是，要训练模型，还需要一个关键信息，即预测是否正确。要了解这一点，将目标 $\boldsymbol{y}_{\text{batch}}$ 的向量与传递给该函数的一系列观测值 $\boldsymbol{X}_{\text{batch}}$ 关联起来，并计算出一个数字，它是 $\boldsymbol{y}_{\text{batch}}$ 和 $\boldsymbol{P}_{\text{batch}}$ 的函数，代表模型对其所做预测的“惩罚”。这里可以选择**均方误差**（mean squared error，MSE），即模型预测“失误”的均方值：

$$\text{MSE}_{(\boldsymbol{P}_{\text{batch}}, \boldsymbol{y}_{\text{batch}})} = \text{MSE}\left(\begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \right) = \frac{(y_1 - P_1)^2 + (y_2 - P_2)^2 + (y_3 - P_3)^2}{3}$$

可以把这个数字称作 L，这里的关键是获得这个数字，一旦有了它，就可以使用第 1 章介绍的所有技术来计算该数字相对于 $\boldsymbol{W}$ 中每个元素的梯度。然后，**可以使用这些导数来更新**

注 7：至少本书中的模型都具有这一特点。

W 的每个元素，逐渐减小 L 的值。可以多次重复这一过程，从而“训练”模型。你将在本章中看到，这样做确实可以在实践中发挥作用。为了清楚地了解如何计算这些梯度，接下来看一下如何用计算图表示线性回归。

2.3.2　线性回归：更有用的示意图和数学

图 2-3 显示了如何使用第 1 章中的示意图来表示线性回归。

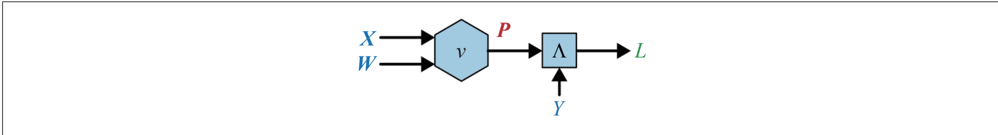


图 2-3：用计算图表示的线性回归方程，其中 X 和 Y 是该函数的数据输入，W 表示权重

为了强调图 2-3 仍然表示一个嵌套函数，可以使用损失值 L，最终计算如下。

$$L = \Lambda(v(\boldsymbol{X}, \boldsymbol{W}), y)$$

2.3.3　加入截距项

将模型表示为示意图，这样做可以从概念上展示如何向模型添加截距项。仅需在末尾添加一个额外的步骤，即添加偏差项，如图 2-4 所示。

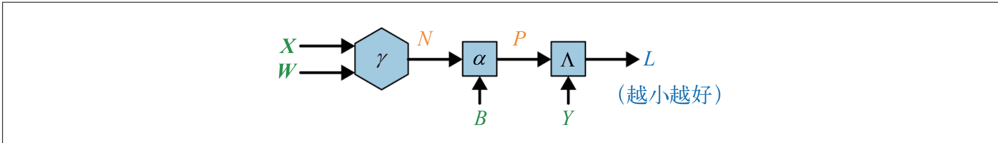


图 2-4：线性回归的计算图，末尾加上一个偏差项

不过在编码之前，我们应该先从数学角度理解原理。当加上偏差项后，模型的预测值 P_i 中的每个元素将是前面所述的点积加上 b：

$$\boldsymbol{P}_{\text{batch_with_bias}} = x_i \text{dot} \boldsymbol{W} + b = \begin{bmatrix} x_{11} \times w_1 + x_{12} \times w_2 + x_{13} \times w_3 + \cdots + x_{1k} \times w_k + b \\ x_{21} \times w_1 + x_{22} \times w_2 + x_{23} \times w_3 + \cdots + x_{2k} \times w_k + b \\ x_{31} \times w_1 + x_{32} \times w_2 + x_{33} \times w_3 + \cdots + x_{3k} \times w_k + b \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

注意，因为线性回归中的截距项是单个数字，对于每个观测值都是一样的，所以应该为传入偏差运算的输入的每个观测值都添加**相同的数字**。本章稍后将讨论这在计算导数方面的意义。

2.3.4　线性回归：代码

现在，将所有内容结合在一起来编写函数，这个函数可以预测，还可以根据给定的观测数据 $\boldsymbol{X}_{\text{batch}}$ 及其对应的目标 $\boldsymbol{y}_{\text{batch}}$ 计算损失值。回想一下，使用链式法则为嵌套函数计算导数涉及两个步骤：首先，执行前向传递，通过一系列运算将输入连续向前传递，并保存计算所得的量；然后，使用这些量在后向传递期间计算导数。

下面的代码执行上述操作，将前向传递中计算所得的量保存在字典中。此外，为了区分前向传递中计算所得的量和参数本身（后向传递中同样需要），该函数将接受一个包含参数的字典：

```
def forward_linear_regression(X_batch: ndarray,
                              y_batch: ndarray,
                              weights: Dict[str, ndarray])
                              -> Tuple[float, Dict[str, ndarray]]:
    '''
    分步线性回归的前向传递。
    '''
    # 断言X和y的批次大小相等
    assert X_batch.shape[0] == y_batch.shape[0]

    # 断言矩阵乘法有效
    assert X_batch.shape[1] == weights['W'].shape[0]

    # 断言B只是1×1的ndarray
    assert weights['B'].shape[0] == weights['B'].shape[1] == 1

    # 执行前向传递中的运算
    N = np.dot(X_batch, weights['W'])

    P = N + weights['B']

    loss = np.mean(np.power(y_batch - P, 2))

    # 保存前向传递中计算的信息
    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X_batch
    forward_info['N'] = N
    forward_info['P'] = P
    forward_info['y'] = y_batch

    return loss, forward_info
```

现在，我们已经准备就绪，可以开始“训练”这个模型了。接下来详细了解训练模型的意义及其实现方法。

2.4　训练模型

现在使用第 1 章介绍的所有工具为 $\boldsymbol{W}$ 中的每个 w_i 计算 $\frac{\partial L}{\partial w_i}$，同时也会计算 $\frac{\partial L}{\partial b}$。如何实现呢？因为该函数的前向传递是通过一系列嵌套函数传递输入的，所以后向传递将仅涉及计

算每个函数的偏导数，算出函数输入处的那些导数并将它们相乘。即使涉及矩阵乘法，也可以使用第 1 章介绍的方法来处理这一问题。

2.4.1　计算梯度：示意图

从概念上讲，我们需要类似图 2-5 所示的效果。

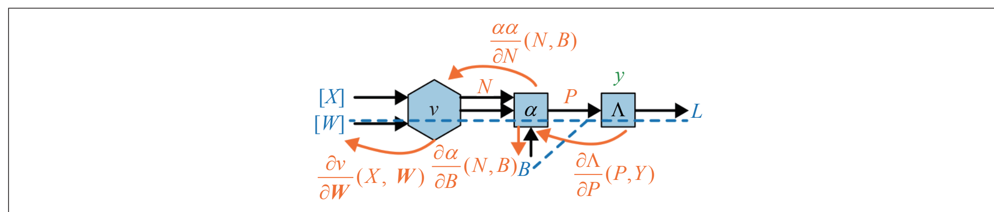


图 2-5：后向传递线性回归计算图

我们只需向后退，计算每个组成函数的导数，算出函数在前向传递中输入处的导数，最后将这些导数相乘。这是大致的过程，下面来看一下实施细节。

2.4.2　计算梯度：数学和一些代码

从图 2-5 可以看到，最终要计算的导数乘积为：

$$\frac{\partial \Lambda}{\partial \boldsymbol{P}}(\boldsymbol{P},Y)\times\frac{\partial \alpha}{\partial N}(N,B)\times\frac{\partial v}{\partial \boldsymbol{W}}(\boldsymbol{X},\boldsymbol{W})$$

它由 3 个部分组成，下面依次计算每个部分。

首先是 $\frac{\partial \Lambda}{\partial \boldsymbol{P}}(\boldsymbol{P},Y)$。对于 Y 和 $\boldsymbol{P}$ 中的每个元素，因为 $\Lambda(\boldsymbol{P},Y)=(Y-\boldsymbol{P})^2$，所以可以得到：

$$\frac{\partial \Lambda}{\partial \boldsymbol{P}}(\boldsymbol{P},Y)=-1\times(2\times(Y-\boldsymbol{P}))$$

我们有点超前了，但代码很简单：

```
dLdP = -2 * (Y - P)
```

接下来要计算的是一个涉及矩阵的表达式：$\frac{\partial \alpha}{\partial N}(N,B)$。由于 α 只是加法，因此在第 1 章中用数字计算的逻辑同样适用于此，那就是将 N 中的任意元素增加 1 单位，那么 $P=\alpha(N,B)=N+B$ 会同时增加 1 单位。这样一来，$\frac{\partial \alpha}{\partial N}(N,B)$ 只是一个 +1+s 的矩阵，其形状与 N 相同。因此，这个表达式对应的代码非常简单，如下所示：

```
dPdN = np.ones_like(N)
```

最后计算 $\frac{\partial v}{\partial \boldsymbol{W}}(\boldsymbol{X},\boldsymbol{W})$。正如第 1 章所述，当计算嵌套函数的导数时（其中一个组成函数是矩阵乘法），可以这样做：

$$\frac{\partial v}{\partial \boldsymbol{W}}(\boldsymbol{X},\boldsymbol{W}) = \boldsymbol{X}^{\mathrm{T}}$$

代码很简单：

```
dNdW = np.transpose(X, (1, 0))
```

下面对截距项执行相同的操作。因为只是将其相加，所以截距项相对于输出的偏导数就是 1：

```
dPdB = np.ones_like(weights['B'])
```

最后一步是将它们相乘，确保根据第 1 章所述的计算过程对涉及 `dNdW` 和 `dNdX` 的矩阵乘法使用正确的顺序。

2.4.3 计算梯度：完整的代码

我们的目标是把所有要计算或输入的内容都进行前向传递（包括图 2-5 中的 $\boldsymbol{X}$、$\boldsymbol{W}$、N、B、$\boldsymbol{P}$ 和 y）并计算 $\frac{\partial \Lambda}{\partial \boldsymbol{W}}$ 和 $\frac{\partial \Lambda}{\partial B}$。下面的代码在名为 `weights` 的字典中接收 $\boldsymbol{W}$ 和 B 作为输入，并在名为 `forward_info` 的字典中接收其余的量：

```
def loss_gradients(forward_info: Dict[str, ndarray],
                   weights: Dict[str, ndarray]) -> Dict[str, ndarray]:
    '''
    为分步线性回归模型计算dLdW和dLdB。
    '''
    batch_size = forward_info['X'].shape[0]

    dLdP = -2 * (forward_info['y'] - forward_info['P'])

    dPdN = np.ones_like(forward_info['N'])

    dPdB = np.ones_like(weights['B'])

    dLdN = dLdP * dPdN

    dNdW = np.transpose(forward_info['X'], (1, 0))

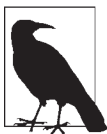
    # 需要在此处使用矩阵乘法，左侧为dNdW（参见第1章末尾的注释）
    dLdW = np.dot(dNdW, dLdN)

    # 需要沿表示批次大小的维度求和（参见本章末尾的注释）
    dLdB = (dLdP * dPdB).sum(axis=0)

    loss_gradients: Dict[str, ndarray] = {}
    loss_gradients['W'] = dLdW
    loss_gradients['B'] = dLdB

    return loss_gradients
```

如你所见，只需计算每个运算的导数，然后将它们依次相乘，注意按正确的顺序进行矩阵乘法[8]。你将在下文中了解到，这种方法确实有效，而且第 1 章已经介绍了链式法则，这样做其实并不奇怪。



关于这些损失梯度，有一个实现细节：将它们存储为字典，其中权重的名称作为键，权重的增加对损失的影响作为值。weights 字典的结构与此相同。因此，我们将按以下方式迭代模型中的权重：

```
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

以这种方式存储它们没有什么特别之处。如果采用不同的存储方式，只需遍历并以不同的方式引用它们即可。

2.4.4 使用梯度训练模型

接下来，只需重复执行以下步骤即可。

1. 选择一批数据。
2. 执行前向传递。
3. 使用在前向传递中计算所得的信息执行后向传递。
4. 使用在后向传递中计算的梯度来更新权重。

在本章的 Jupyter Notebook 中有一个 train 函数，该函数对以上过程进行了编码，过程没有太大新意，只是实现了上述步骤，并添加了一些合理的内容，例如混洗数据——确保数据以随机顺序输入。以下是关键的代码行，它们会在 for 循环内重复出现：

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

for key in weights.keys(): # weights和loss_grads具有相同的键
    weights[key] -= learning_rate * loss_grads[key]
```

然后，把 train 函数运行一定的轮数，或者在整个训练数据集中循环运行它，如下所示：

```
train_info = train(X_train, y_train,
                   learning_rate = 0.001,
                   batch_size=23,
                   return_weights=True,
                   seed=80718)
```

train 函数返回 train_info 元组，其中一个元素是表示模型所学内容的参数，也就是权重。

注 8：另外，必须沿轴 0 对 dLdB 求和，本章稍后会涉及这一步骤的更多细节。



在深度学习中，“参数”和“权重”这两个术语可以互换。鉴于此，本书将灵活地使用它们。

2.5　评估模型：训练集与测试集

要判断模型是否揭示了数据间的关系，必须从统计学中引入一些术语和思维方式。我们认为所有接收到的数据集都是来自**总体**（population）的**样本**（sample）。尽管有时仅有一个样本，但目标始终是找到揭示总体中各种关系的模型。

这里始终存在一种风险，即构建的模型拾取了样本中存在但总体中并不存在的关系。例如，在示例中可能是带有 3 间浴室的黄色石板房相对便宜，虽然总体中可能不存在这种关系，但我们构建的复杂神经网络模型很可能采用这种关系。这是一个**过拟合**（overfitting）问题。如何检测所使用的模型结构是否可能出现这个问题呢？

解决方案就是将样本分为**训练集**（training set）和**测试集**（testing set）。使用训练集训练模型（迭代更新权重），然后在测试集上评估模型的性能。

这里的逻辑在于，如果模型能够成功地**从训练集泛化到样本的其余部分（整个数据集）**，那么相同的“模型结构”将很可能**从样本（整个数据集）泛化到总体**，这也是最终目标。

2.6　评估模型：代码

基于前述内容，下面在测试集上评估模型。首先，我们将编写一个函数，通过截断 `forward_loss` 函数来生成预测结果：

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]):
    '''
    从分步线性回归模型生成预测结果。
    '''
    N = np.dot(X, weights['W'])

    return N + weights['B']
```

然后，使用之前从 `train` 函数返回的权重编写以下代码：

```
preds = predict(X_test, weights) # weights = train_info[0]
```

预测准确度如何呢？这里采用看似奇怪的方法，将模型定义为一系列运算，并通过迭代调整相关参数来训练它们。在调整参数时，需要计算相对于使用链式法则的参数的损失偏导数。但要注意的是，现在这种方法还没有得到验证。如果这种方法行之有效，一定会很有帮助。

为了查看模型是否有效，可以绘制图表，用 x 轴表示模型的预测值，用 y 轴表示实际值。如果每个点正好落在 45 度线上，则该模型将是完美的。图 2-6 显示了模型的预测值和实际值的曲线图。

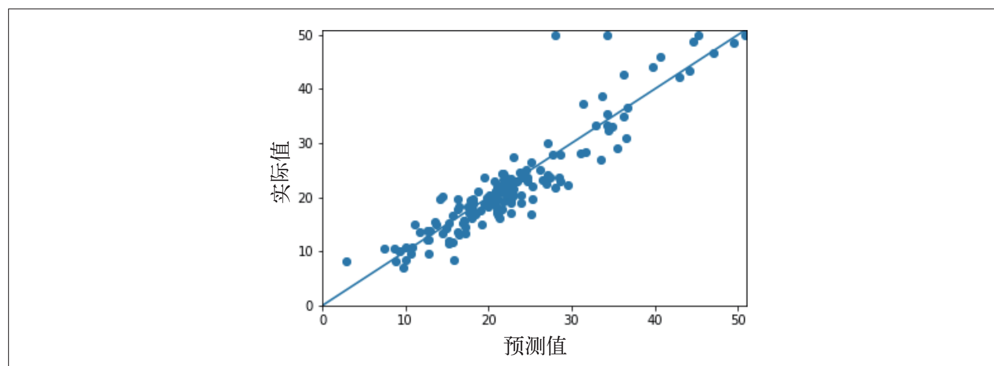


图 2-6：自定义线性回归模型的预测值与实际值

图 2-6 看起来不错，现在来量化模型的好坏。有许多常见的方法可以做到这一点，举例如下。

- 计算模型的预测值与实际值之间的平均距离（以绝对值表示），该度量称为**平均绝对误差**（mean absolute error）：

```
def mae(preds: ndarray, actuals: ndarray):
    '''
    计算平均绝对误差。
    '''
    return np.mean(np.abs(preds - actuals))
```

- 计算模型的预测值与实际值之间的均方距离，该度量称为**均方根误差**（root mean squared error）：

```
def rmse(preds: ndarray, actuals: ndarray):
    '''
    计算均方根误差。
    '''
    return np.sqrt(np.mean(np.power(preds - actuals, 2)))
```

对本例中的模型来说，平均绝对误差是 3.5643，均方根误差是 5.0508。

由于和目标的标度相同，因此均方根误差是一种特别常见的度量。如果将这个数字除以目标的平均值，就可以得出预测值与实际值相差的平均度量。由于 `y_test` 的平均值为 22.0776，因此该模型对房价的预测平均降低了 5.0508/22.076 ≅ 22.9%。

这些数字有什么用吗？在包含本章代码的 Jupyter Notebook 中，可以看到，当使用最流行的 Python 机器学习库 scikit-learn 对该数据集执行线性回归时，得到的平均绝对误差和

均方根误差分别为 3.5666 和 5.0482，这实际上与之前在“基于基本原则”的线性回归中计算的结果相差无几。这也在一定程度上证明，本书所采用的方法确实是推算和训练的有效方法！在本章的后面部分和第 3 章中，我们会将这种方法扩展到神经网络和深度学习模型中。

分析最重要的特征

在开始建模之前，缩放数据的每个特征，使其均值为 0，标准差为 1。这样做在计算上具有优势，第 4 章将详细讨论。另外，针对线性回归，这种做法的一个好处是可以将系数的绝对值解释为不同特征对模型的重要性。系数越大，意味着特征越重要。系数如下所示：

```
np.round(weights['W'].reshape(-1), 4)

array([-1.0084, 0.7097, 0.2731, 0.7161, -2.2163, 2.3737, 0.7156,
       -2.6609, 2.629, -1.8113, -2.3347, 0.8541, -4.2003]
```

最后一个系数的绝对值最大，这意味着数据集中的最后一个特征最为重要。图 2-7 针对目标值绘制了这个特征。

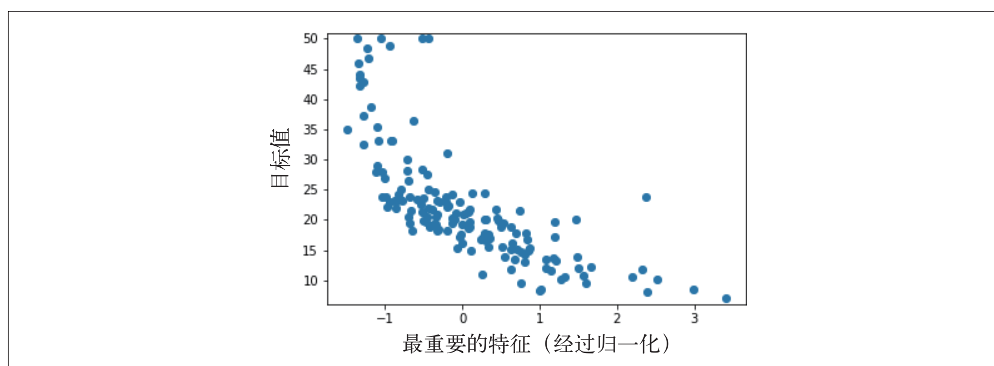


图 2-7：自定义线性回归中最重要的特征与目标值

可以看到，这个特征确实与目标值密切相关：随着该特征的增加，目标值逐渐减小，反之亦然。但是，这种关系不是线性的。当特征从 –2 更改为 –1 时，目标值更改的预期量与特征从 1 更改为 2 时目标值更改的预期量**不同**。稍后再来看这一点。

图 2-8 相比图 2-7 有一些变化，那就是将这个特征和**模型预测值**之间的关系叠加在一起。接下来通过把以下数据输入到训练模型中来实现这一点。

- 所有特征的值均等于其平均值。
- 针对最重要的特征，分 40 步进行线性插值（从 –1.5 到 3.5），这大约是数据中该特征的缩放范围。

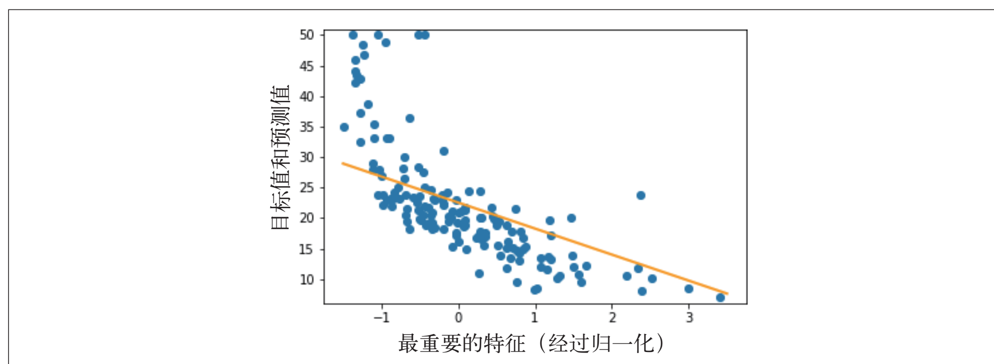


图 2-8：自定义线性回归模型中最重要的特征与目标值和模型预测值的对比

图 2-8 显示了线性回归模型的局限性：尽管事实上该特征与目标之间存在明显且“可模型化”的非线性关系，但由于它的内在结构，模型只能“学习”线性关系。

为了使模型学习特征与目标之间更复杂的非线性关系，必须构建一个比线性回归更复杂的模型。但是怎么实现呢？接下来就来看一种解决方案：根据基本原理构建神经网络。

2.7 从零开始构建神经网络

我们已经了解了如何根据基本原理构建和训练线性回归模型。如何将这种计算链扩展到可以学习非线性关系的更复杂的模型中呢？核心思想就是，首先执行一系列线性回归，然后将结果输入给一个非线性函数，最终执行最后一个线性回归并做出预测。事实证明，可以为这个更复杂的模型计算梯度，就像对线性回归模型所做的那样。

2.7.1 步骤1：一系列线性回归

执行“一系列线性回归”是什么意思呢？执行一次线性回归涉及对一组参数进行矩阵乘法：如果数据 X 的维度为 [batch_size， num_features]，那么将其乘以维度为 [num_features， 1] 的权重矩阵 W，最终将得到维度为 [batch_size， 1] 的输出。对于批次中的每个观测值，该输出只是原始特征的**加权总和**。要执行多元线性回归，只需将输入乘以维度为 [num_features， num_outputs] 的权重矩阵，即可得到维度为 [batch_size， num_outputs] 的输出。现在，**对于每个观测值**，都有原始特征的 num_outputs 个加权和。

可以将这些加权和视为“已学习到的特征”，即原始特征的组合。以预测房价为例，一旦神经网络得到训练，它将试图学习有助于准确预测房价的特征组合。应该创建多少个已学习到的特征呢？答案是创建 13 个，对应所创建的 13 个原始特征。

2.7.2 步骤2：一个非线性函数

接下来把每个加权和输入到一个非线性函数中。我们要尝试的第一个函数是第 1 章提到的 sigmoid 函数。先来回顾一下 sigmoid 函数，如图 2-9 所示。

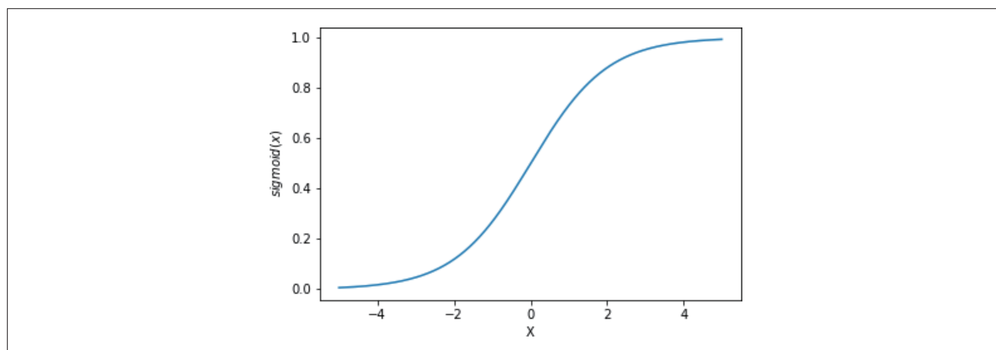


图 2-9：sigmoid 函数，x 的取值范围是 –5 ～ 5

为什么使用非线性函数（sigmoid 函数），而不是平方函数（ $f(x)=x^2$ ）或其他类型的函数呢？主要涉及以下几个原因。第一，我们希望此处使用的函数是**单调**的，以便它“保留”有关输入数字的信息。假设给定输入的日期，两次线性回归得到的值分别为 –3 和 3，那么将这些数字输入到平方函数中，得到的值都是 9。因此，在将这些数字输入平方函数之后，任何接收它们作为输入的函数都将“丢失”信息，即不知道具体输入的是 –3 还是 3。

第二，这个函数是非线性的，这种非线性将使神经网络能够对特征与目标之间固有的非线性关系进行建模。

第三，sigmoid 函数有一个很好的属性，即它的导数可以用函数本身来表示：

$$\frac{\partial \sigma}{\partial u}(x)=\sigma(x)\times(1-\sigma(x))$$

当在神经网络的后向传递中使用 sigmoid 函数时，就会利用这一点。

2.7.3 步骤3：另一个线性回归

最后将得到 13 个元素，其中每个元素都是原始特征的组合，当把它们输入到 sigmoid 函数后，取值都在 0 和 1 之间。同时，将这些元素输入到一个常规的线性回归模型中，使用它们的方式与之前使用原始特征的方式相同。

尝试用本章前面训练标准线性回归模型的方法来训练整个结果函数：将数据输入到模型中，使用链式法则计算增加权重会增加（或减少）损失的程度，然后在每次迭代中按照减少损失的方向更新权重。随着时间的推移，在理想状态下，最终会得到一个比以前更精确

的模型，而且该模型已经“学到”了特征和目标之间的内在非线性关系。

根据这一描述，你可能很难搞清楚具体的细节，示意图可能会有所帮助。

2.7.4　示意图

至此，我们已经有了一个比较复杂的模型，其计算图如图 2-10 所示。

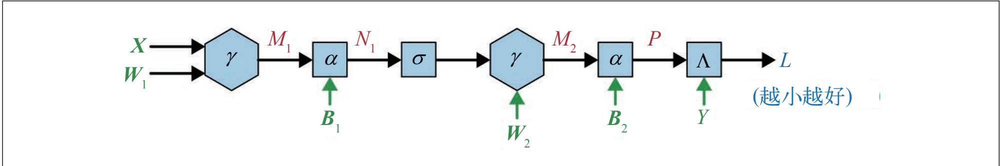


图 2-10：将 3 个步骤转换为计算图

可以看到，这里和以前一样，还是从矩阵乘法和矩阵加法开始。现在将前面提到的一些术语标准化：在嵌套函数中应用这些运算时，用于转换输入特征的第一个矩阵叫作**权重矩阵**，第二个矩阵（添加到每组结果特征集的元素）叫作**偏差矩阵**。在图 2-10 中，它们分别表示为 $\boldsymbol{W}_1$ 和 $\boldsymbol{B}_1$。

在应用这些运算之后，我们将结果输入到 sigmoid 函数中，然后用另一组权重矩阵和偏差矩阵（分别称为 $\boldsymbol{W}_2$ 和 $\boldsymbol{B}_2$）再次重复这个过程，从而得到最终的预测值 P。

另一种示意图？

用单独的步骤来表示计算过程，这样做是否有助于直观地理解呢？这个问题是本书的主题。要完全理解神经网络，必须看到多种表示形式，其中每种表示形式都突出了神经网络工作原理的不同方面。另外，虽然图 2-10 中的表示形式并没有针对网络结构给出太多的信息，但它确实清楚地表明了这种模型的训练方法：在后向传递时，计算每个组成函数的偏导数，在该函数的输入处进行求值，然后通过将所有导数相乘来计算相对于每个权重的损失梯度，就像第 1 章在链式法则示例中描述的那样。

尽管如此，还有另一种更标准的方式来表示神经网络。首先，可以将每个原始特征表示为一个圆。由于有 13 个特征，因此需要 13 个圆。然后，再增加 13 个圆，用于表示正在执行的“线性回归 sigmoid”运算的 13 个输出。此外，由于所有这些圆都是包含 13 个原始特征的函数，因此需要将第一组中的 13 个圆分别连接到第二组中的所有圆[9]。最后，因为这 13 个输出将全部用于最终预测，所以再画一个圆来表示最终预测结果，并用 13 条线表示“中间输出”已经连接到了最终预测结果。图 2-11 展示了最终的示意图[10]。

注 9：有趣的是，可以只将输出连接到某些原始特征。这实际上就是卷积神经网络所做的事情。

注 10：其实，这并非“最终”的示意图——在前两层特征之间连线的话，共需 169 条线，这里虽然没有全部画出这 169 条线，但足以解释这个概念。

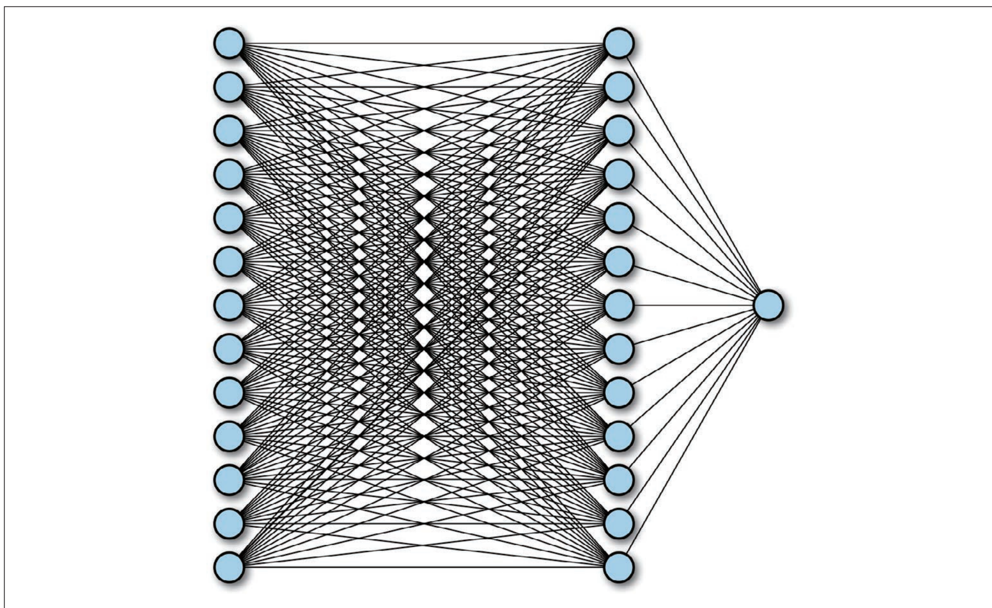


图 2-11：较为常见但作用稍逊的神经网络表示法

如果以前阅读过有关神经网络的内容，那么你不会对图 2-11 所示的表示法感到陌生。尽管这种表示法有一定的优势，能够非常清楚地展示神经网络的类型、层级数量等，但它既没有给出任何实际计算所涉及的信息，也没有说明如何训练这样一个神经网络。因此，虽然这种示意图非常重要（在其他地方也会出现），但它不是本书的重点。我之所以在这里展示它，是为了说明它与本书所用的表示法有何联系。本书采用盒子表示法，每个盒子代表一个函数，该函数定义了模型在前向传递和后向传递中的运行情况，包括在前向传递中哪些需要预测，在后向传递中哪些需要学习。第 3 章会介绍如何将每个函数编码为从 `Operation` 基类继承的 Python 类，从而在示意图和代码之间进行更直接的转换。

2.7.5　代码

本章已经介绍了如何编写简单的线性回归函数，在对神经网络进行编码时，需要遵循与之相同的函数结构，将 `weights` 作为字典，同时返回损失值和 `forward_info` 字典，并在内部使用图 2-10 指定的运算进行替换：

```
def forward_loss(X: ndarray,
                 y: ndarray,
                 weights: Dict[str, ndarray]
                 ) -> Tuple[Dict[str, ndarray], float]:
    '''
    计算分步神经网络模型的前向传递结果和损失值。
    '''
```

```python
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']

    loss = np.mean(np.power(y - P, 2))

    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X
    forward_info['M1'] = M1
    forward_info['N1'] = N1
    forward_info['O1'] = O1
    forward_info['M2'] = M2
    forward_info['P'] = P
    forward_info['y'] = y

    return forward_info, loss
```

尽管现在的示意图较为复杂，但是我们仍然分步完成了每一个运算，并将结果保存在 `forward_info` 中。

2.7.6 神经网络：后向传递

在神经网络中，后向传递的原理与在简单的线性回归模型中的原理相同，只不过步骤相对多一点。

1. 示意图

这里主要涉及 2 个步骤。

(1) 计算每个运算的导数，并在其输入处进行求值。
(2) 将结果相乘。

根据链式法则，这里执行上述步骤依然有效。图 2-12 展示了需要计算的所有偏导数。

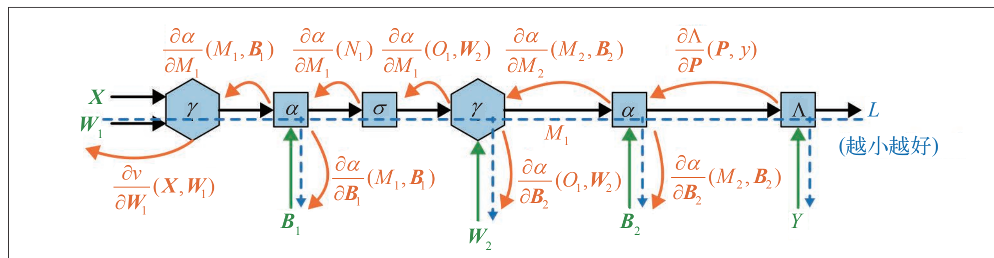


图 2-12：与神经网络中的每个运算都相关的偏导数，这些偏导数将在后向传递时相乘

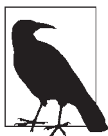
就像在线性回归模型中执行的运算一样，这里要计算所有这些偏导数，对函数进行后向跟踪，然后将它们相乘，得到每个权重的损失梯度。

2. 数学和代码

表 2-1 列出了图 2-12 中的偏导数及其对应的代码行。

表2-1：神经网络偏导数表

偏　导　数	代　　码
$\frac{\partial \Lambda}{\partial P}(P, y)$	`dLdP = -(forward_info[y] - forward_info[P])`
$\frac{\partial \alpha}{\partial M_2}(M_2, B_2)$	`np.ones_like(forward_info[M2])`
$\frac{\partial \alpha}{\partial B_2}(M_2, B_2)$	`np.ones_like(weights[B2])`
$\frac{\partial \alpha}{\partial M_1}(O_1, W_2)$	`dM2dW2 = np.transpose(forward_info[O1], (1, 0))`
$\frac{\partial \alpha}{\partial B_2}(O_1, W_2)$	`dM2dO1 = np.transpose(weights[W2], (1, 0))`
$\frac{\partial \alpha}{\partial M_1}(N_1)$	`dO1dN1 = sigmoid(forward_info[N1] × (1 - sigmoid(forward_info[N1])`
$\frac{\partial \alpha}{\partial M_1}(M_1, B_1)$	`dN1dM1 = np.ones_like(forward_info[M1])`
$\frac{\partial \alpha}{\partial B_1}(M_1, B_1)$	`dN1dB1 = np.ones_like(weights[B1])`
$\frac{\partial v}{\partial W_1}(X, W_1)$	`dM1dW1 = np.transpose(forward_info[X], (1, 0))`



用于计算偏差项 dLdB1 和 dLdB2 的损失梯度表达式必须沿行进行求和，这样表明在传递的数据批次中，每行都添加了相同的偏差元素。参见附录中的“关于偏差项的损失梯度”。

3. 总损失梯度

可以在本章的 Jupyter Notebook 中找到完整的 loss_gradients 函数。该函数计算表 2-1 中的每一个偏导数，并将它们相乘来获得相对于每个权重 ndarray 的损失梯度。

- dLdW2
- dLdB2
- dLdW1
- dLdB1

唯一需要注意的是，如附录中的“关于偏差项的损失梯度”一节所述，对于沿轴 0 计算的偏导数（dLdB1 和 dLdB2），这里对其表达式进行求和。

我们终于从零开始构建了第一个神经网络！接下来看一下实际的运行情况，并判断它是否比线性回归模型更为有效。

2.8 训练和评估第一个神经网络

与本章已经构建的线性回归模型一样，前向传递和后向传递对神经网络起到了相同的作用，训练方法和评估方法也类似：对于数据的每次迭代，通过前向传递将输入传递至函数，通过后向传递计算相对于权重的损失梯度，然后使用这些梯度更新权重。实际上，可以在训练循环中使用以下代码：

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

更新后的区别仅在于 forward_loss 函数和 loss_gradients 函数的内部以及 weights 字典，后者现在有 4 个键（W1、B1、W2 和 B2），而不是 2 个。实际上，这是本书的一个重要主题：即使模型的架构非常复杂，其数学原理和总体的训练过程也与简单模型是一样的。

以同样的方式从这个模型中得到预测结果：

```
preds = predict(X_test, weights)
```

同样，区别只在于 predict 函数的内部：

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]) -> ndarray:
    '''
    从分步神经网络模型生成预测结果。
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']

    return P
```

使用这些预测结果，可以像以前一样计算验证集上的平均绝对误差和均方根误差，二者分

别为 2.5289 和 3.6775。

与之前的模型相比，该模型的这两个值都明显较低！从图 2-13 中预测值与目标值的对比可以看出类似的改进结果。

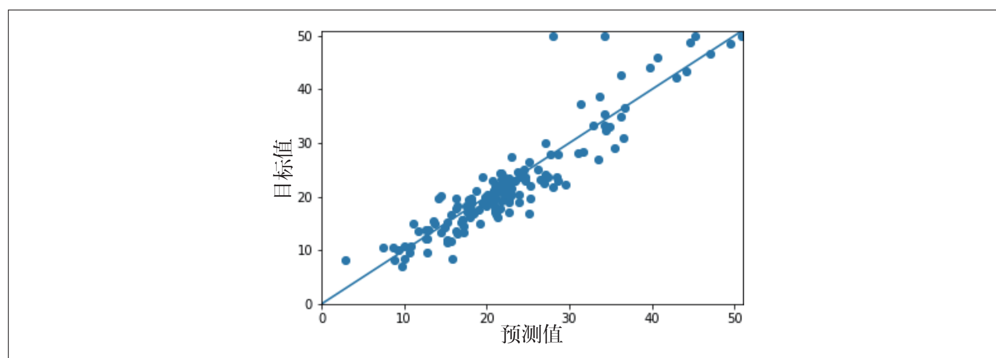


图 2-13：神经网络回归模型的预测值与目标值

推荐逐步跟随本章的 Jupyter Notebook 运行代码。

发生这种情况的两个原因

为什么神经网络模型看起来比之前的模型效果更好呢？回想一下，对于之前的线性回归模型，在其最重要的特征与目标之间存在**非线性关系**，但模型只能学习单个特征与目标之间的**线性关系**。本书提到，通过加入一个非线性函数，可以让模型正确地学习特征和目标之间的非线性关系。

下面通过图表来说明。与在线性回归模型中一样，我们来看看神经网络回归模型中最重要的特征与目标值和模型预测值的关系，如图 2-14 所示。像以前一样，最重要的特征的取值范围是 –1.5 ～ 3.5。

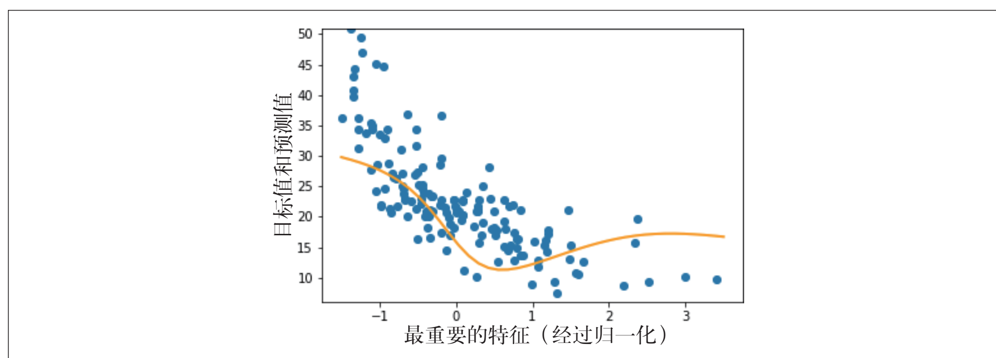


图 2-14：神经网络回归模型中最重要的特征与目标值和模型预测值的对比

从图 2-14 中可以看到以下两点：第一，其中所示的关系是非线性的；第二，和我们所希望的一样，图中的关系更接近于特征与目标（由点表示）之间的关系。因此，将非线性函数添加到模型中可以使它通过训练迭代更新权重，从而学习输入和输出之间的非线性关系。

以上就是神经网络比简单的线性回归模型效果更好的第一个原因。第二个原因是，除了单个特征，神经网络还可以学习**原始特征**与目标之间的**组合关系**。这是因为神经网络使用矩阵乘法来创建多个“已学习到的特征”，每个特征都是所有原始特征的组合，本质上相当于在这些“已学习到的特征”上应用另一个线性回归。例如，通过一些探索性分析，可以看到该模型所学到的 13 个原始特征的两个最重要的组合：

$$-4.44 \times feature_6 - 2.77 \times feature_1 - 2.07 \times feature_7 + \cdots$$
$$4.43 \times feature_2 - 3.39 \times feature_4 - 2.39 \times feature_1 + \cdots$$

然后，这些特征将与所学到的其他 11 个特征一起，包含在神经网络最后两层的线性回归中。

学习单个特征与目标之间的非线性关系，以及学习特征与目标之间的组合关系，这两点决定了在解决实际问题时，神经网络通常比简单的线性回归模型更为有效。

2.9 小结

本章介绍了如何使用第 1 章中的构成要素和思维模型来理解、构建和训练两个标准的机器学习模型，从而解决实际问题。首先，本章展示了如何使用计算图来表示基于经典统计学（线性回归）的简单机器学习模型。这种表示方法能够计算出这个模型相对于其参数的损失梯度，从而通过不断地输入训练集中的数据，并沿减少损失的方向更新模型参数，来训练模型。

接着，本章介绍了该模型的局限性：只能学习特征和目标之间的线性关系。因为需要学习特征与目标之间的非线性关系，所以本章构建了第一个神经网络。我们了解了如何从零开始构建神经网络，并且学习了如何使用与训练线性回归模型相同的过程对其进行训练。最后，根据经验判断，可以发现神经网络的性能优于简单的线性回归模型，并了解了两个关键原因，即神经网络既能够学习特征与目标之间的非线性关系，也能够学习特征与目标之间的组合关系。

当然，本章构建的神经网络仍然相对简单，这是因为用本章所述方式定义神经网络是一个极费精力的过程。定义前向传递涉及 6 个需要独立编码的运算，定义后向传递则涉及 17 个。但是，敏锐的读者会注意到，这些步骤有很多重复之处，而且通过正确定义抽象，可以从根据单个运算来定义模型（如本章所述）转向根据这些抽象来定义模型。这样便能够构建更复杂的模型（包括深度学习模型），同时加深对这些模型工作原理的理解。这就是第 3 章要介绍的内容。加油！

第 3 章

从零开始深度学习

你可能没有意识到，自己现在掌握的数学知识和概念足以回答本书开头提出的有关深度学习模型的关键问题。你了解了神经网络的**工作原理**，即涉及矩阵乘法、损失值以及与该损失值相关的偏导数的计算，也了解了这些计算能起作用的原因，即微积分的链式法则。这些内容可以通过遵循基本原则构建神经网络来理解，也就是将它们表示为一系列构成要素，其中每个构成要素都是单独的数学函数。本章介绍如何将这些构成要素本身表示为抽象的 Python 类，然后使用这些类构建深度学习模型。学完本章，你将真正完成“从零开始深度学习”！

本章还会根据上述构成要素，将神经网络的描述映射到关于深度学习模型的更常规的描述，而且你可能对这些描述并不陌生。举例来说，学完本章，你将了解深度学习模型具有“多个隐藏层”的含义。这实际上是理解概念的本质：能够在总体描述和底层细节之间自由转换。下面便开始着手转换。前两章仅涉及根据发生的底层运算来描述模型，本章将把对模型的描述映射到如“层”等常见的高级概念，这些概念最终有助于轻松地描述更复杂的模型。

3.1　定义深度学习

深度学习模型是什么呢？前文将“模型”定义为由计算图表示的数学函数。这种模型的目的是将来自某个数据集且具有共同特征的输入（例如代表房屋特征的输入）映射到从相关分布中提取的输出（例如房屋的价格）。在这一过程中可以发现，如果将模型定义为一个函数，其中包含一些参数（在该函数的某些运算中作为输入），则可以通过以下步骤对其进行“拟合”，从而以最佳的方式描述数据。

1. 反复给模型输入观测值，跟踪在前向传递过程中计算所得的量。
2. 计算**损失值**，它表示模型的预测值与目标值之间的差距。
3. 使用在前向传递过程中计算所得的量和第 1 章得出的数学链式法则，计算每个输入参数对该损失值造成的最终影响。
4. 更新参数值。这样一来，当下一组观测值通过模型时，便有机会减少损失。

回顾第 2 章，我们首先构建了仅包含一系列线性运算的模型，将特征转换为目标（结果证明，这相当于构建传统的线性回归模型）。但是，该模型具有明显的局限性：即使能够“完美地拟合”，它也只能表示特征和目标之间的线性关系。

然后，我们定义了一个函数结构，该函数结构首先应用上述线性运算，接着应用**非线性运算**（sigmoid 函数），最后应用一组线性运算。可以看到，通过这种修改，模型学到的输入和输出之间的关系更接近真实的非线性关系。同时，这种模型具有额外的优势，那就是可以学习输入特征与目标之间的**组合关系**。

像这样的模型和深度学习模型之间有什么联系呢？尝试理解一个定义：深度学习模型表示为一系列运算，这些运算至少涉及两个不连续的非线性函数。

要注意的是，由于深度学习模型只是一系列运算，因此其训练过程实际上与简单模型的训练过程**相同**。毕竟，训练过程之所以有效，就是因为模型相对于其输入具有可微性。如第 1 章所述，可微函数的组合也是可微的。因此，只要组成函数的各个运算是可微的，整个函数就是可微的，就可以用上述 4 个步骤来训练模型。

但是，目前训练这些模型的方法实际上是通过手动编码前向传递和后向传递，然后将适当的量相乘来计算导数。第 2 章中的简单神经网络模型共需要 17 步，由于这是在低层次上描述模型，因此尚不清楚如何提高该模型的复杂度，甚至不知道如何实现简单的改变，比如用另一个非线性函数替换 sigmoid 函数。为了能够构建任意“深度”和“复杂度”的深度学习模型，在这 17 个步骤中，必须考虑创建可重用组件。这些可重用组件的级别要高于单个运算，这样可以相互进行替换，从而构建不同的模型。神经网络往往被描述为由“层”“神经元”等构成。为了朝着正确的方向创建抽象，可以尝试把运算与“层”“神经元”等传统描述联系起来。

第一步必须创建一个抽象来表示目前正在使用的各个运算，而不是继续重复编写相同的矩阵乘法和加法运算。

3.2　神经网络的构成要素：运算

我们用 `Operation` 类代表神经网络中的组成函数。根据此类函数在模型中的使用方式，它应该具有 `forward` 方法和 `backward` 方法，每个方法都接受一个 `ndarray` 作为输入，并输出一个 `ndarray`。某些运算（例如矩阵乘法）似乎有另一种特殊的输入，即代表参数的 `ndarray`。

在 `Operation` 类或继承它的其他类中，可以把 `params` 作为另一个实例变量。

可以将 `Operation` 类分为两种类型：某些 `Operation` 类（例如矩阵乘法）返回 `ndarray` 作为输出，其形状与作为输入的 `ndarray` 不同；某些 `Operation` 类（例如 sigmoid 函数）则仅将某个函数应用于输入 `ndarray` 中的所有元素。对于在运算之间传递的 `ndarray` 的形状，其遵循怎样的“一般规则”呢？可以考虑在 `Operation` 类中传递的 `ndarray`：每个 `Operation` 类将在前向传递中向前发送输出，并在后向传递中接收“输出梯度”，该梯度表示相对于 `Operation` 类输出的每个元素的损失偏导数，该数值由组成神经网络的其他 `Operation` 类计算得出。同样，在后向传递中，每个 `Operation` 类将向后发送“输入梯度”，该梯度表示相对于输入的每个元素的损失偏导数。

上述事实对 `Operation` 类的运作设置了一些重要限制，有助于确保正确地计算梯度，其中主要涉及以下两个方面。

1. **输出梯度** `ndarray` 的形状必须与**输出**的形状相匹配。
2. 对于 `Operation` 类在后向传递期间向后发送的**输入梯度**，其形状必须与**输入**的形状相匹配。

当用示意图解释时，这一切都会变得更清楚。

3.2.1 示意图

在图 3-1 中，运算 `O` 从运算 `N` 中接收输入并将输出传递给运算 `P`。

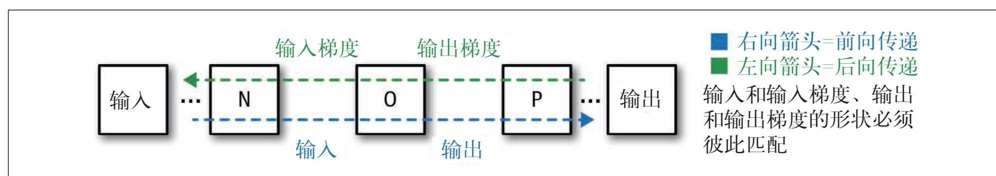


图 3-1：具有输入和输出的运算

图 3-2 展示了运算带有参数的情况。

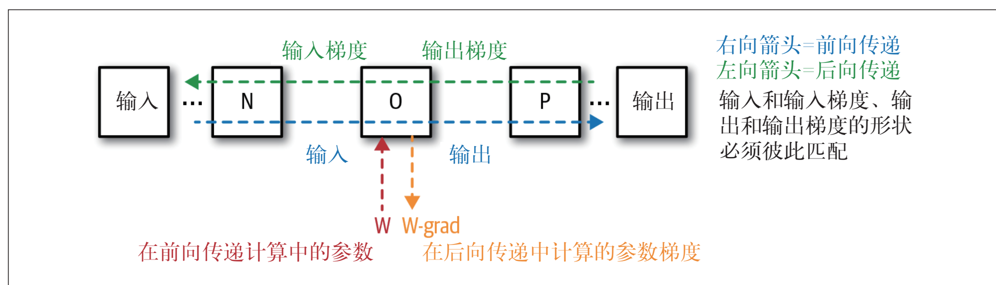


图 3-2：具有输入和输出以及参数的运算

3.2.2 代码

根据前面的介绍，可以为神经网络编写基本的构成要素，即 Operation 类，如下所示：

```
class Operation(object):
    '''
    Operation基类。
    '''
    def __init__(self):
        pass

    def forward(self, input_: ndarray):
        '''
        将输入存储在self.input_实例变量中，调用self._output函数。
        '''
        self.input_ = input_

        self.output = self._output()

        return self.output


    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        调用self._input_grad函数，检查形状是否匹配。
        '''
        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        return self.input_grad


    def _output(self) -> ndarray:
        '''
        必须为每个Operation类都定义_output方法。
        '''
        raise NotImplementedError()


    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        必须为每个Operation类都定义_input_grad方法。
        '''
        raise NotImplementedError()
```

对于定义的任意 Operation 类，都必须实现 _output 方法和 _input_grad 方法。之所以这样命名，主要是为了体现它们计算的量。



像上面这样定义基类主要是出于教学目的：请记住，深度学习中的所有 Operation 类都要向前发送输出和向后发送梯度。另外，Operation 类在前向传递中接收的形状必须与在后向传递中发送的形状相匹配，反之亦然。

本章在后面将定义矩阵乘法等具体的 `Operation` 类。首先来定义另一个类，它继承 `Operation` 类，并且专门用于涉及参数的运算：

```
class ParamOperation(Operation):
    '''
    带有参数的Operation类。
    '''

    def __init__(self, param: ndarray) -> ndarray:
        '''
        ParamOperation方法。
        '''
        super().__init__()
        self.param = param

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        调用self._input_grad函数和self._param_grad函数，检查形状是否匹配。
        '''

        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)
        self.param_grad = self._param_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        assert_same_shape(self.param, self.param_grad)

        return self.input_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        ParamOperation的每个子类都必须实现_param_grad方法。
        '''
        raise NotImplementedError()
```

与 `Operation` 基类类似，除了 `_output` 方法和 `_input_grad` 方法，单个 `ParamOperation` 类还必须定义 `_param_grad` 方法。

至此，我们已经完成了对 `Operation` 类的定义，但在定义神经网络之前，还要定义另一个构成要素。

3.3 神经网络的构成要素：层

就 `Operation` 类而言，层是一系列线性运算外加后面跟着的一个非线性运算。举例来说，第 2 章中的神经网络共有 5 个运算：2 个线性运算（加权乘法和偏差项加法）、1 个非线性运算（sigmoid 函数），以及 2 个线性运算。在这种情况下，可以说前 3 个运算（包括非线性运算在内）构成第 1 层，后 2 个运算构成第 2 层。另外，输入本身代表一种特殊的层，称为**输入层**。（就层的编号而言，由于该层不计算在内，因此可以将其视为第 0 层。）同

理，最后一层称为**输出层**。中间层（根据编号为第 1 层）也有一个重要的名称，它被称为**隐藏层**，这是因为它的值在训练过程中通常是隐藏的。

从层的上述定义来看，输出层是一个例外——它无须应用非线性运算。这仅仅是因为，我们经常希望输出层的取值范围是 $-\infty \sim \infty$（或至少是 $0 \sim \infty$），而非线性函数通常将其输出“压缩”到某个更小的取值范围。例如，sigmoid 函数将其输出的取值范围压缩为 $0 \sim 1$。

示意图

为了使“层”的概念更加清晰，图 3-3 将第 2 章中的神经网络示意图划分为 3 层。

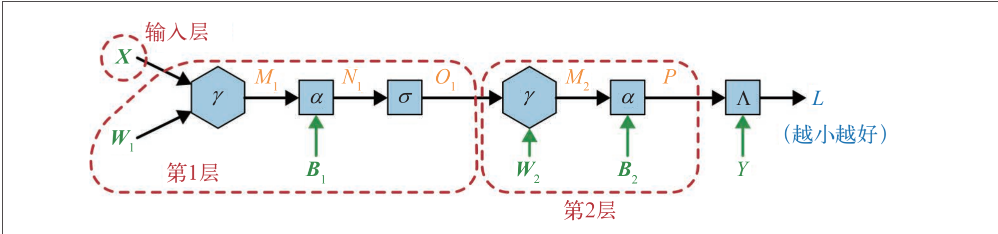


图 3-3：将神经网络示意图中的运算分为若干层

可以看到，输入就代表输入层，其后的 3 个运算（以 sigmoid 函数结尾）代表第 1 层，最后的两个运算则代表第 2 层。

当然，这样表示起来很麻烦。对不止两层的神经网络而言，将其表示为一系列单独的运算，同时清楚地显示其工作方式以及训练方式，这样做过于细化了。这就是要用“层”来表示神经网络的原因，如图 3-4 所示。

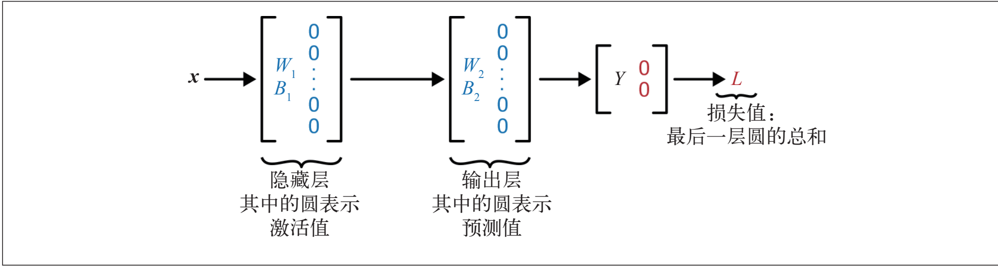


图 3-4：用“层”来表示神经网络

与脑神经网络的联系

现在，将上述内容与你可能听说过的一个概念联系起来：神经网络的每一层都具有一定数量的**神经元**（neuron），这些神经元的数量等于表示**该层输出中每个观测值向量的维数**。因此，可以认为先前示例中的神经网络在输入层具有 13 个神经元，在隐藏层同样具有 13 个神经元，但在输出层具有 1 个神经元。

大脑中的神经元具有接收来自许多其他神经元输入的特性，并且只有当所接收的信号累积达到一定的“激活能量”时，它们才会发送信号。人工神经网络中的神经元具有类似的属性：它们确实根据输入来向前发送信号，但是输入只通过一个非线性函数转换成输出。该非线性函数称为**激活函数**（activation function），由此产生的值称为该层的**激活值**（activation）[1]。

既然已经定义了层，现在就可以陈述更为传统的深度学习定义：**深度学习模型是具有多个隐藏层的神经网络。**

可以看到，这与前文给出的仅使用运算描述的定义一致，因为一个层就是一系列运算，其中最后一个是非线性运算。

既然已经为 Operation 类定义了基类，那么接下来看一下它如何充当模型的基本构成要素。

3.4　在构成要素之上构建新的要素

为了使第 2 章中的模型生效，需要实现哪些 Operation 类呢？根据分步实现该神经网络的经验，可以判断存在以下 3 种 Operation 类。

- 输入与参数矩阵的矩阵乘法。
- 增加一个偏差项。
- sigmoid 激活函数。

下面从第 1 种 Operation 类开始实现，我们将它命名为 WeightMultiply：

```
class WeightMultiply(ParamOperation):
    '''
    神经网络的权重乘法运算。
    '''

    def __init__(self, W: ndarray):
        '''
        使用self.param = W初始化Operation类。
        '''
        super().__init__(W)

    def _output(self) -> ndarray:
        '''
        计算输出。
        '''
        return np.dot(self.input_, self.param)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        计算输入梯度。
        '''
```

注 1：在所有激活函数中，sigmoid 函数（将输入映射到 0 和 1 之间）最大限度地模拟了大脑神经元的实际激活机制，但一般来说，激活函数可以是任何单调的非线性函数。

```
        return np.dot(output_grad, np.transpose(self.param, (1, 0)))

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        计算参数梯度。
        '''
        return np.dot(np.transpose(self.input_, (1, 0)), output_grad)
```

这里只需使用第 1 章提到的计算规则对两部分进行编码：第一，前向传递上的矩阵乘法；第二，后向传递上的输入和参数的“向后发送梯度”的规则。这部分稍后将进行展示，现在可以把它用作一种**构成要素**，并简单地将其插入到 Layer 类中。

接下来实现加法运算，可以称之为 BiasAdd：

```
class BiasAdd(ParamOperation):
    '''
    增加偏差项。
    '''

    def __init__(self,
                 B: ndarray):
        '''
        使用self.param = B初始化Operation类。检查形状。
        '''
        assert B.shape[0] == 1

        super().__init__(B)

    def _output(self) -> ndarray:
        '''
        计算输出。
        '''
        return self.input_ + self.param

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        计算输入梯度。
        '''
        return np.ones_like(self.input_) * output_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        计算参数梯度。
        '''
        param_grad = np.ones_like(self.param) * output_grad
        return np.sum(param_grad, axis=0).reshape(1, param_grad.shape[1])
```

最后，定义 Sigmoid 类（sigmoid 激活函数）：

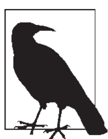
```
class Sigmoid(Operation):
    '''
    sigmoid激活函数。
    '''
```

```
    def __init__(self) -> None:
        '''不做处理。'''
        super().__init__()

    def _output(self) -> ndarray:
        '''
        计算输出。
        '''
        return 1.0/(1.0+np.exp(-1.0 * self.input_))

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        计算输入梯度。
        '''
        sigmoid_backward = self.output * (1.0 - self.output)
        input_grad = sigmoid_backward * output_grad
        return input_grad
```

这里仅实现了第 2 章描述的数学函数。



对于 Sigmoid 和 ParamOperation，在后向传递步骤中的计算为：input_grad = < 某一项 > * output_grad。我们在该步骤中应用链式法则。正如第 1 章所述，当涉及矩阵乘法时，针对 WeightMultiply 的相应规则与链式法则是类似的。

在精确定义了这些 Operation 类后，下面可以将它们用作定义 Layer 类的构成要素。

3.4.1 层的蓝图

得益于 Operation 类的编写方式，Layer 类编写起来很容易，其中涉及下面两个步骤。

1. 就像本书一直在示意图中所展示的那样，forward 方法和 backward 方法只涉及通过一系列 Operation 类连续向前发送输入。这是有关 Layer 类工作机制的最重要的事实。代码的其余部分是对此机制的包装，主要涉及以下操作。
   (1) 在 _setup_layer 函数中定义一系列 Operation 类，并在这些 Operation 类中初始化和存储参数（_setup_layer 函数也将执行该操作）。
   (2) 在 forward 方法中将正确的值存储在 self.input_ 和 self.output 中。
   (3) 在 backward 方法中执行正确的断言检查。
2. _params 函数和 _param_grads 函数仅从层内的 ParamOperation 中提取参数及其梯度（相对于损失）。

整个 Layer 类看起来像这样：

```
class Layer(object):
    '''
    神经网络中的一个神经元层。
    '''
```

```
def __init__(self,
             neurons: int):
        '''
        神经元的数量大致对应该层的“宽度”。
        '''
        self.neurons = neurons
        self.first = True
        self.params: List[ndarray] = []
        self.param_grads: List[ndarray] = []
        self.operations: List[Operation] = []

    def _setup_layer(self, num_in: int) -> None:
        '''
        必须为每一层都实现_setup_layer函数。
        '''
        raise NotImplementedError()

    def forward(self, input_: ndarray) -> ndarray:
        '''
        通过一系列Operation类将输入向前传递。
        '''
        if self.first:
            self._setup_layer(input_)
            self.first = False

        self.input_ = input_

        for operation in self.operations:

            input_ = operation.forward(input_)

        self.output = input_

        return self.output

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        通过一系列Operation类将output_grad向后传递，检查形状。
        '''
        assert_same_shape(self.output, output_grad)

        for operation in reversed(self.operations):
            output_grad = operation.backward(output_grad)

        input_grad = output_grad

        self._param_grads()

        return input_grad

    def _param_grads(self) -> ndarray:
        '''
        从层的运算中提取参数梯度。
        '''
```

```
        self.param_grads = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.param_grads.append(operation.param_grad)

    def _params(self) -> ndarray:
        '''
        从层的运算中提取参数。
        '''

        self.params = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.params.append(operation.param)
```

就像在第 2 章中先定义 Operation 类再针对神经网络具体实现一样，接下来为神经网络实现 Layer 类。

3.4.2 稠密层

我们把 Operation 类称为 WeightMultiply 和 BiasAdd 等。对于 Layer 类，应该如何称呼呢？ LinearNonLinear 层？

神经元层的一个关键特征是，**每个输出神经元都是所有输入神经元的函数**。这就是矩阵乘法的实际作用：如果矩阵有 n_{in} 行和 n_{out} 列，那 么乘法本身就是在计算 n_{out} 个新特征，每一个新特征都是所有 n_{in} 个输入特征的加权线性组合 [2]。因此，这些层通常称为**全连接层**（fully connected layer）。在流行的 Keras 库中，它们也被称为 Dense 层，这是一个含义相同但更简洁的术语。

在了解它的名称及其由来之后，下面根据已经定义的运算来定义 Dense 层。正如你将看到的，基于定义 Layer 基类的方法，这里只需将上一节中定义的 Operation 类作为一个列表放在 _setup_layer 函数中即可。

```
class Dense(Layer):
    '''
    从Layer类继承全连接层。
    '''
    def __init__(self,
                 neurons: int,
                 activation: Operation = Sigmoid()) -> None:
        '''
        在初始化时需要一个激活函数。
        '''
        super().__init__(neurons)
```

注 2：正如将在第 5 章中看到的，并非所有层都是这样。举例来说，在卷积层中，每个输出特征都只是一小部分输入特征的组合。

```python
        self.activation = activation

    def _setup_layer(self, input_: ndarray) -> None:
        '''
        定义全连接层的运算。
        '''
        if self.seed:
            np.random.seed(self.seed)

        self.params = []

        # 权重
        self.params.append(np.random.randn(input_.shape[1], self.neurons))

        # 偏差
        self.params.append(np.random.randn(1, self.neurons))

        self.operations = [WeightMultiply(self.params[0]),
                           BiasAdd(self.params[1]),
                           self.activation]

        return None
```

注意，这里将默认激活设置为线性激活，这实际上意味着不应用任何激活，只是将同一函数应用于层的输出。

现在，应该在 `Operation` 类和 `Layer` 类上添加哪些构成要素呢？为了训练模型，需要一个 `NeuralNetwork` 类来包装 `Layer` 类，就像 `Layer` 类包装 `Operation` 类一样。由于目前还不清楚是否需要更多的类，因此我们将深入研究并构建 `NeuralNetwork` 类，并在构建过程中找出所需要的其他类。

3.5 NeuralNetwork类和其他类

总体而言，`NeuralNetwork` 类应该能够**从数据中学习**，更准确地说，它应该能够获取表示“观测值”（X）和“正确答案”（y）的批数据，并学习 X 和 y 之间的关系。这意味着学习一个函数，该函数可以将 X 转换为非常接近 y 的预测值 `prediction`。

基于已经定义的 `Layer` 类和 `Operation` 类，这种学习将如何进行呢？回顾第 2 章中的模型，这里将实现以下 3 项内容。

1. 神经网络应获取 X，将其连续向前传递给每个 `Layer` 类（实际上是将 X 传递给许多 `Operation` 类的便捷包装器），此时的 `prediction` 将代表结果。
2. 将 `prediction` 与值 y 进行比较，来计算损失并生成“损失梯度”，也就是损失相对于神经网络最后一层（生成 `prediction` 的层）中的每个元素的偏导数。
3. 使用计算“参数梯度”（损失相对于每个参数的偏导数）的方式，将该损失梯度依次向后传递给各层，并将其存储在相应的 `Operation` 类中。

3.5.1 示意图

图 3-5 展示了以 Layer 类的形式描述神经网络。

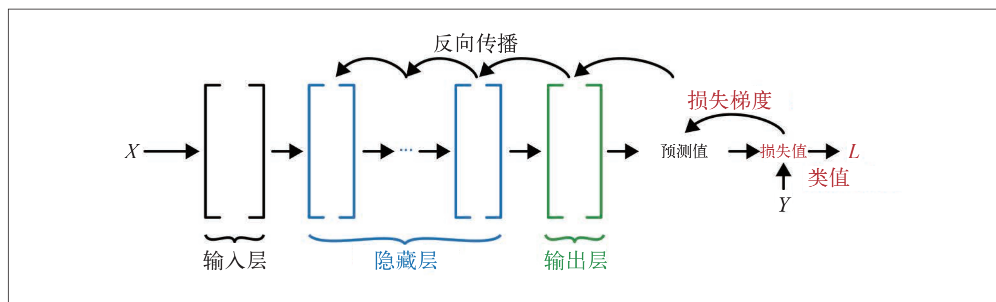


图 3-5：反向传播（以层的概念展示）

3.5.2 代码

应该如何实现呢？当神经网络处理 Layer 类时，最终采用的方式要与 Layer 类处理 Operation 类的方式相同。例如，forward 方法接受 X 作为输入，并简单地执行如下操作：

```
for layer in self.layers:
    X = layer.forward(X)

return X
```

类似地，backward 方法也可以接受一个参数（暂且将其称为 grad），然后执行如下操作：

```
for layer in reversed(self.layers):
    grad = layer.backward(grad)
```

grad 必须来自损失函数，这种特殊的函数同时接受 y 和 prediction，然后执行如下操作。

1. 计算一个数字，表示对得出该 prediction 的神经网络的“惩罚”。
2. 对于损失的每个 prediction，向后发送梯度。这个梯度将作为该神经网络的 backward 方法的输入，由最后一个 Layer 类接收。

在第 2 章的示例中，损失函数是 prediction 与目标值之间的平方差，可以据此计算 prediction 相对于损失的梯度。

这里应该如何实现呢？“损失”这个概念似乎很重要，值得拥有自己的类。此外，这个类的实现类似于 Layer 类，但有一个例外，那就是 forward 方法将生成一个实际的数字（float 类型）作为损失，而不是生成要发送到下一个 Layer 类的 ndarray。接下来看看如何正式实现。

3.5.3 Loss类

类似于 Layer 类，Loss 类的 forward 方法和 backward 方法将检查相应 ndarray 的形状是否相同，并定义 _output 方法和 _input_grad 方法，Loss 类的所有子类都必须实现这两个方法：

```
class Loss(object):
    '''
    神经网络的"损失"。
    '''

    def __init__(self):
        '''不做处理。'''
        pass

    def forward(self, prediction: ndarray, target: ndarray) -> float:
        '''
        计算实际的损失值。
        '''
        assert_same_shape(prediction, target)

        self.prediction = prediction
        self.target = target

        loss_value = self._output()

        return loss_value

    def backward(self) -> ndarray:
        '''
        计算损失值相对于损失函数输入的梯度。
        '''
        self.input_grad = self._input_grad()

        assert_same_shape(self.prediction, self.input_grad)

        return self.input_grad

    def _output(self) -> float:
        '''
        Loss类的每个子类都必须实现_output方法。
        '''
        raise NotImplementedError()

    def _input_grad(self) -> ndarray:
        '''
        Loss类的每个子类都必须实现_input_grad方法。
        '''
        raise NotImplementedError()
```

与 Operation 类一样，这里检查损失后向传递的梯度与从神经网络最后一层输入的 prediction 形状是否相同：

```
class MeanSquaredError(Loss):

    def __init__(self)
        '''不做处理。'''
        super().__init__()

    def _output(self) -> float:
        '''
        计算每个观测值的平方误差损失。
        '''
        loss =
            np.sum(np.power(self.prediction - self.target, 2)) /
            self.prediction.shape[0]

        return loss

    def _input_grad(self) -> ndarray:
        '''
        在MSE损失函数中，计算相对于输入的损失梯度。
        '''

        return 2.0 * (self.prediction - self.target) / self.prediction.shape[0]
```

这里对均方误差损失公式的前向规则和后向规则简单地进行了编码。

这是需要从零开始构建深度学习模型的最后一个关键构成要素。接下来回顾所有构成要素以及它们的组合方式，然后正式构建模型！

3.6 从零开始构建深度学习模型

我们将以图 3-5 为指导，来构建 NeuralNetwork 类，并最终使用该类来定义和训练深度学习模型。在深入研究并开始编码之前，先来精确地描述这样一个类，以及它如何与前面定义的 Operation 类、Layer 类和 Loss 类进行交互。

1. NeuralNetwork 类将拥有一系列 Layer 类作为属性。与前面定义的一样，这些 Layer 类将具有 forward 方法和 backward 方法。这些方法接受并返回 ndarray。
2. 在 _setup_layer 函数中，每个 Layer 类都将在该层的 operations 属性中保存一个由 Operation 类组成的列表。
3. 这些 Operation 类与 Layer 类本身一样，具有 forward 方法和 backward 方法，这些方法接受 ndarray 作为参数，并返回 ndarray 作为输出。
4. 在每个 Operation 类中，backward 方法接收的 output_grad 的形状必须与 Layer 类的 output 属性的形状相同。对于 backward 方法和 input_ 属性中向后传递的 input_grad 的形状也是如此。
5. 一些运算具有参数（存储在 param 属性中），这些运算继承自 ParamOperation 类。针对输入形状和输出形状的约束，同样适用于 Layer 类及其 forward 方法和 backward 方法，它们接受并输出 ndarray。另外，输入属性和输出属性及其对应的梯度在形状上必须匹配。

6. NeuralNetwork 类也会有一个 Loss 类。该类将获取最后一次运算的输出及目标，检查它们的形状是否相同，并计算损失值（一个数字）和一个 loss_grad ndarray，这些值将被传递给输出层，从而启动反向传播。

3.6.1 实现批量训练

在训练模型时，第 2 章介绍了一次训练一批数据的高级步骤。它们很重要，可以再重新看一下。

1. 将输入传递给模型函数（“前向传递”），获得一个预测值。
2. 计算表示损失的数字。
3. 使用链式法则和前向传递过程中所计算的量，计算相对于参数的损失梯度。
4. 使用这些梯度更新参数。

在完成上述步骤之后，输入一批新数据并重复这些步骤。

将这些步骤转换为上述 NeuralNetwork 框架很简单，涉及下面 5 个步骤。

1. 接受 X 和 y 作为输入，两者都是 ndarray。
2. 将 X 依次向前传递给每个 Layer 类。
3. 使用 Loss 类生成损失值，并将损失梯度向后发送。
4. 使用损失梯度作为神经网络 backward 方法的输入，该方法将为神经网络中的每一层都计算参数梯度。
5. 在每一层上都调用 update_params 函数，该函数将使用 NeuralNetwork 类的整体学习率以及新计算的参数梯度。

至此，我们终于完整地定义了一个能够适应批量训练的神经网络，接下来对其进行编码。

3.6.2 NeuralNetwork:代码

对神经网络进行编码非常简单：

```
class NeuralNetwork(object):
    '''
    神经网络对应的类。
    '''
    def __init__(self, layers: List[Layer],
                 loss: Loss,
                 seed: float = 1)
        '''
        神经网络需要层和一个损失变量。
        '''
        self.layers = layers
        self.loss = loss
        self.seed = seed
```

```
        if seed:
            for layer in self.layers:
                setattr(layer, "seed", self.seed)

    def forward(self, x_batch: ndarray) -> ndarray:
        '''
        将数据向前传递给一系列层。
        '''
        x_out = x_batch
        for layer in self.layers:
            x_out = layer.forward(x_out)

        return x_out

    def backward(self, loss_grad: ndarray) -> None:
        '''
        将数据向后传递给一系列层。
        '''

        grad = loss_grad
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

        return None

    def train_batch(self,
                    x_batch: ndarray,
                    y_batch: ndarray) -> float:
        '''
        将数据向前传递给各层。计算损失。将数据向后传递给各层。
        '''

        predictions = self.forward(x_batch)

        loss = self.loss.forward(predictions, y_batch)

        self.backward(self.loss.backward())

        return loss

    def params(self):
        '''
        获取神经网络的参数。
        '''
        for layer in self.layers:
            yield from layer.params

    def param_grads(self):
        '''
        获取相对于神经网络参数的损失梯度。
        '''
        for layer in self.layers:
            yield from layer.param_grads
```

借助这个 NeuralNetwork 类，可以用更模块化、更灵活的方式实现第 2 章中的模型，并定义其他模型来表示输入和输出之间的复杂非线性关系。例如，下面展示了如何轻松实例化第 2 章介绍的两个模型，即线性回归模型和神经网络[3]：

```
linear_regression = NeuralNetwork(
    layers=[Dense(neurons = 1)],
            loss = MeanSquaredError(),
            learning_rate = 0.01
            )

neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=Linear())],
            loss = MeanSquaredError(),
            learning_rate = 0.01
            )
```

以上已经基本完成了编码工作。现在，只需要向模型反复输入数据，以便让其进行学习。为了使该过程更清晰、更容易扩展到更复杂的深度学习场景（参见第 4 章），可以定义另一个执行训练的类以及一个附加类，让附加类执行“学习”任务，或者根据后向传递上计算的梯度对 NeuralNetwork 类进行更新，这种做法很有帮助。下面来快速定义这两个类。

3.7 优化器和训练器

在第 2 章中，我们使用了以下代码来实现训练模型的 4 个步骤：

```
# 向前传递X_batch并计算损失
forward_info, loss = forward_loss(X_batch, y_batch, weights)

# 计算每个权重的损失梯度
loss_grads = loss_gradients(forward_info, weights)

# 更新权重
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

这些代码位于 for 循环中，该循环通过函数定义反复输入数据并更新了神经网络。

对于现有的类，最终将在 Trainer 类的 fit 函数中完成这项操作，该函数主要是第 2 章使用的 train 函数的包装器。主要区别在于，在这个新函数中，代码块的前两行将被替换为以下代码行：

```
neural_network.train_batch(X_batch, y_batch)
```

注 3：0.01 的学习率没有特殊含义，只是在写第 2 章的时候，我发现它在实验过程中是最优的。

在接下来的两行中发生的参数更新将在单独的 `Optimizer` 类中进行。最后，之前包装了所有这些的 `for` 循环将在 `Trainer` 类中进行，该类包装了 `NeuralNetwork` 类和 `Optimizer` 类。

接下来介绍需要 `Optimizer` 类的原因，以及它应该具备的形态。

3.7.1 优化器

在第 2 章描述的模型中，每个 `Layer` 类都包含一个简单的规则，用于根据参数及其梯度来更新权重。还可以使用许多其他的更新规则（参见第 4 章），例如涉及梯度更新历史的相关规则，而不仅仅是在这个迭代中输入的特定批次的梯度更新。也可以创建一个单独的 `Optimizer` 类（优化器），这样在替换更新规则时将更为灵活，第 4 章将对此进行更详细的探讨。

说明和代码

`Optimizer` 类将包含一个 `NeuralNetwork` 类，并且每次调用 `step` 函数时，都会基于它们的当前值、梯度以及存储在 `Optimizer` 类中的其他任意信息来更新神经网络的参数：

```
class Optimizer(object):
    '''
    神经网络优化器基类。
    '''
    def __init__(self,
                 lr: float = 0.01):
        '''
        每个优化器都必须具有初始学习率。
        '''
        self.lr = lr

    def step(self) -> None:
        '''
        每个优化器都必须实现step函数。
        '''
        pass
```

以下应用简单的更新规则，即**随机梯度下降**（stochastic gradient descent）：

```
class SGD(Optimizer):
    '''
    随机梯度下降优化器。
    '''
    def __init__(self,
                 lr: float = 0.01) -> None:
        '''不做处理。'''
        super().__init__(lr)

    def step(self):
        '''
        对于每个参数，都按照合适的方向进行调整，调整的幅度基于学习率。
        '''
```

```
        for (param, param_grad) in zip(self.net.params(),
                                       self.net.param_grads()):

            param -= self.lr * param_grad
```



注意，虽然 NeuralNetwork 类没有 _update_params 方法，但确实依赖 params 函数和 param_grads 函数来提取正确的 ndarray 以进行优化。

以上是基本的 Optimizer 类。接下来介绍 Trainer 类。

3.7.2 训练器

除了训练模型，Trainer 类（训练器）还将 NeuralNetwork 类与 Optimizer 类链接在一起，确保后者正确训练前者。你可能已经注意到，3.7.1 节在初始化 Optimizer 类时没有传入 NeuralNetwork 类。相反，稍后当初始化 Trainer 类时，会将 NeuralNetwork 类指定为 Optimizer 类的一个属性，如下所示：

```
setattr(self.optim, 'net', self.net)
```

下面将展示一个简化但有效的 Trainer 类，它目前只包含 fit 函数。该方法对模型进行多轮训练，并在每轮训练后打印出损失值。每轮训练都执行下面两个操作。

1. 在新一轮训练开始时混洗数据。
2. 将数据分批输入到神经网络中，在每批数据传送完成后更新参数。

当将整个训练集输入给 Trainer 类之后，一轮就会结束。

训练器代码

以下代码展示了简单版本的 Trainer 类，我们隐藏了 fit 函数中使用的两个自解释的辅助方法：一个是 generate_batches 方法，它根据 X_train 和 y_train 生成批量数据用于训练；另一个是 permute_data 方法，它在每轮训练开始时对 X_train 和 y_train 进行混洗。train 函数还包含 restart 参数，如果该参数为 True（默认值），那么在调用 train 函数时，它会将模型参数重新初始化为随机值：

```
class Trainer(object):
    '''
    训练神经网络。
    '''
    def __init__(self,
                 net: NeuralNetwork,
                 optim: Optimizer)
        '''
        需要一个神经网络和一个优化器，以进行训练。将神经网络作为实例变量分配给优化器。
        '''
```

```python
        self.net = net
        setattr(self.optim, 'net', self.net)

    def fit(self, X_train: ndarray, y_train: ndarray,
            X_test: ndarray, y_test: ndarray,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32,
            seed: int = 1,
            restart: bool = True) -> None:
        '''
        经过一定轮数的训练，将神经网络与训练数据拟合。对于每eval_every轮，
        它都基于测试数据对神经网络进行计算。
        '''
        np.random.seed(seed)

        if restart:
            for layer in self.net.layers:
                layer.first = True

        for e in range(epochs):

            X_train, y_train = permute_data(X_train, y_train)

            batch_generator = self.generate_batches(X_train, y_train,
                                                    batch_size)

            for ii, (X_batch, y_batch) in enumerate(batch_generator):

                self.net.train_batch(X_batch, y_batch)

                self.optim.step()

            if (e+1) % eval_every == 0:

                test_preds = self.net.forward(X_test)

                loss = self.net.loss.forward(test_preds, y_test)

                print(f"Validation loss after {e+1} epochs is {loss:.3f}")
```

你可以在本书的随书文件中[4]找到完整版本，其中还实现了**提前停止**（early stopping）功能，该功能将执行以下操作。

1. 每 eval_every 轮保存损失值。
2. 检查验证损失是否低于上次计算的损失值。
3. 如果验证损失不低于上次计算的损失值，那么使用 eval_every 轮之前的模型。

当完成上述准备后，便拥有了训练模型所需的一切！

注 4：可以从图灵社区下载随书文件：ituring.cn/book/2759。——编者注

3.8 整合

以下给出使用 Trainer 类和 Optimizer 类以及之前定义的两个模型（linear_regression 和 neural_network）来训练神经网络的完整代码。将学习率设置为 0.01，最大轮数设置为 50，并且每 10 轮评估一次模型：

```
optimizer = SGD(lr=0.01)
trainer = Trainer(linear_regression, optimizer)

trainer.fit(X_train, y_train, X_test, y_test,
            epochs = 50,
            eval_every = 10,
            seed=20190501);

Validation loss after 10 epochs is 30.295
Validation loss after 20 epochs is 28.462
Validation loss after 30 epochs is 26.299
Validation loss after 40 epochs is 25.548
Validation loss after 50 epochs is 25.092
```

使用与第 2 章相同的模型评分函数，并将其包装在 eval_regression_model 函数中，可以得到以下结果：

```
eval_regression_model(linear_regression, X_test, y_test)

Mean absolute error: 3.52

Root mean squared error 5.01
```

可以看到，平均绝对误差为 3.52，均方根误差为 5.01。这些结果与第 2 章中的线性回归结果接近，证明框架是有效的。

使用带有 13 个隐藏神经元的 neural_network 模型运行相同的代码，可以得到以下结果：

```
Validation loss after 10 epochs is 27.434
Validation loss after 20 epochs is 21.834
Validation loss after 30 epochs is 18.915
Validation loss after 40 epochs is 17.193
Validation loss after 50 epochs is 16.214

eval_regression_model(neural_network, X_test, y_test)

Mean absolute error: 2.60

Root mean squared error 4.03
```

同样，这些结果与第 2 章中的结果相似，它们明显优于简单的线性回归模型。

第一个深度学习模型

既然所有这些设置都已完成，那么定义第一个深度学习模型就很简单了：

```
deep_neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=LinearAct())],
    loss=MeanSquaredError(),
    learning_rate=0.01
)
```

我们甚至不必对此多做思考，只需添加一个与第一层具有相同维度的隐藏层。这样一来，该神经网络就有两个隐藏层，每个隐藏层具有 13 个神经元。

使用与先前模型相同的学习率和评估进度进行训练，可以得到以下结果：

```
Validation loss after 10 epochs is 44.134
Validation loss after 20 epochs is 25.271
Validation loss after 30 epochs is 22.341
Validation loss after 40 epochs is 16.464
Validation loss after 50 epochs is 14.604

eval_regression_model(deep_neural_network, X_test, y_test)

Mean absolute error: 2.45

Root mean squared error 3.82
```

我们终于实现了从零开始构建深度学习模型。事实上，在这个实际问题中，除了需要稍微调整学习率，无须其他任何技巧。从这方面来看，深度学习模型确实比只有一个隐藏层的神经网络表现稍好。

更重要的是，这一点可以通过建立易于扩展的框架来实现。假设其他类型的 `Operation` 类已经定义了 `_output` 方法和 `_input_grad` 方法，并且输入、输出和参数的维度与它们各自梯度的维度相匹配，那么就可以轻松实现这些类，并将它们包装在一个新的 `Layer` 中直接使用。同样，还可以轻松地将不同的激活函数放到现有的层中，看看它是否会降低错误指标。我建议你动手尝试一下！

3.9 小结与展望

本章涉及的问题相对简单。第 4 章将介绍一些技术，这些技术对于在更具挑战性的问题中正确训练模型至关重要，尤其是定义其他 `Loss` 类和 `Optimizer` 类。第 4 章还会涉及其他技术，可用于调整学习率并在整个训练过程中对其进行修改，同时将演示如何在 `Optimizer` 类和 `Trainer` 类中应用这些技术。最后会介绍 dropout，这是一种新的 `Operation` 类，对提高深度学习模型的训练稳定性至关重要。加油！

第4章 扩展

第 1 章和第 2 章从基本原则出发，探讨了深度学习模型的概念及其工作原理。在此基础上，第 3 章构建了第一个深度学习模型，并对其进行了训练，旨在解决相对简单的房价预测问题。但是，在大多数现实问题中，成功训练深度学习模型并不是那么容易。尽管可以想象这些模型能够找到任意问题（监督学习问题）的最优解决方案，但事实上它们经常会失败，而且对于给定的模型架构，很难在理论上保证为给定的问题找到良好的解决方案。尽管如此，还是可以应用一些技术，这些技术易于理解，而且能提高神经网络训练成功的概率。本章的重点就是这些技术。

本章首先回顾神经网络在数学上“尝试做的事情”，即找到函数的最小值，其次展示一系列有助于神经网络实现这一点的技术，并在经典的 MNIST 手写数字数据集上证明其有效性。我们将从损失函数开始学习，该函数应用于深度学习的整个分类问题。你将看到，它能够显著地加速学习过程[1]。此外，我们还会讨论除 sigmoid 之外的其他激活函数，这些激活函数也可以加速学习过程。接着介绍**动量**（momentum），这是随机梯度下降优化技术的最重要、最直接的扩展，同时简要讨论更高级的优化器。最后介绍 3 种互不相关但必不可少的技术：学习率衰减、权重初始化和 dropout。正如你将看到的，每种技术都将帮助神经网络依次找到更优的解决方案。

第 1 章遵循“数学、示意图、代码”的结构介绍了每个概念。在本章中，由于每种技术都没有明确的示意图，因此这里将从关于每种技术的“直觉”开始，然后进行数学运算（通常比第 1 章中的内容要简单得多），最后以代码结束。这实际上需要将技术整合到已经构

注 1：目前本书只讨论了回归问题，由于还没有引入这个损失函数，因此还不能合理地处理分类问题。

建的框架中，从而精确地描述这些技术如何与第 3 章介绍的构成要素进行交互。基于这个思路，本章从神经网络的目标讲起，即找到函数的最小值。

4.1 关于神经网络的一些直觉

如前所述，神经网络包含许多权重。有了这些权重，再加上一些输入数据 X 和 y，就可以计算出最终的损失值。图 4-1 粗略地展示了神经网络，尽管十分简化，但这种描绘方式仍然正确。

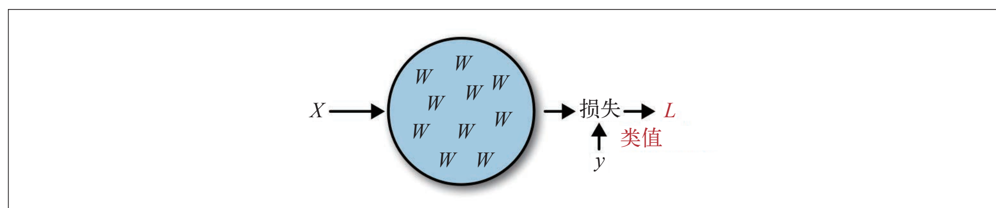


图 4-1：以简化的方式描绘带有权重的神经网络

实际上，每一个单独的权重都与特征 X、目标 y、其他权重以及最终的损失值 L 有着复杂的非线性关系。如果在保持其他权重、X 和 y 的值不变的情况下改变权重 W，并绘制出损失值 L 的变化情况，那么可以看到如图 4-2 所示的结果。

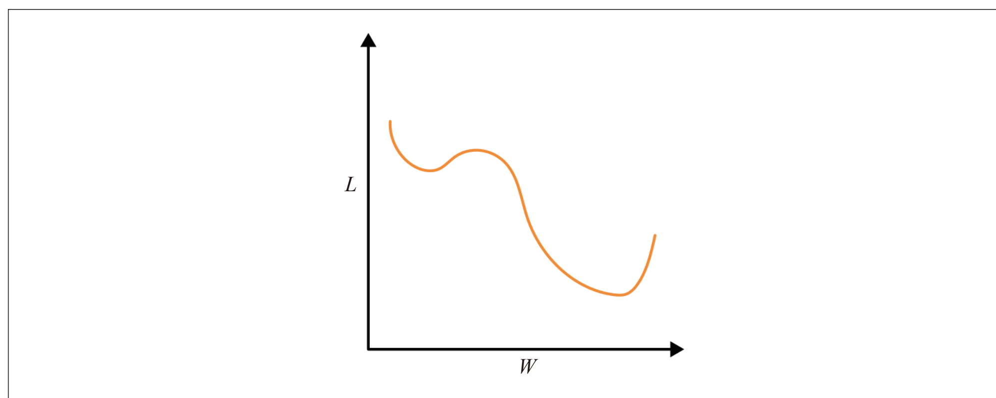


图 4-2：神经网络的权重与损失

当开始训练神经网络时，首先将每个权重初始化为图 4-2 中横轴上的某个值。然后，使用在反向传播过程中计算出的梯度，迭代更新权重，并根据该曲线在选择的初始值处的斜率进行第一次更新[2]。图 4-3 从几何角度解释了基于梯度和学习率更新神经网络权重的含义。左边部分的箭头表示重复地应用更新规则，其学习率比右边部分的箭头小。注意，在这两

注 2：此外，正如你在第 3 章中所看到的，将这些梯度乘以学习率，以便对权重的变化进行更精细的控制。

种情况下，水平方向上的更新都与权重值处曲线的斜率成比例（斜率越大表示更新程度越大）。

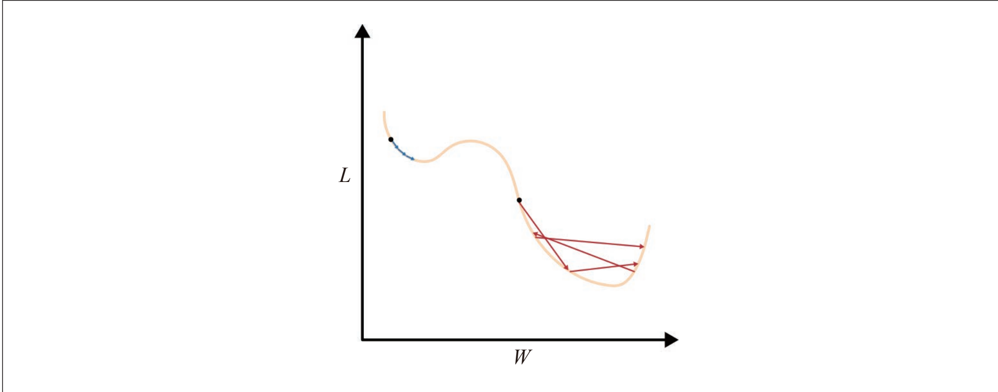


图 4-3：以几何方式描绘如何根据梯度和学习率更新神经网络的权重

训练深度学习模型的目标是将每个权重都移动到使损失最小化的“全局”值。从图 4-3 中可以看出，如果采取的步长太小，就有可能最终达到“局部”最小值，这要次于全局最小值（左边部分的箭头展示了遵循该方案的权重路径）。如果步长太大，即使靠近了全局最小值，也有可能“反复跳过”它（如右边部分的箭头所示）。这是在调整学习率时要权衡的根本问题：如果学习率太小，就可能会陷入局部最小值；如果学习率太大，则可能跳过全局最小值。

实际上，情况要比这复杂得多。原因之一是，神经网络具有成千上万个甚至数百万个权重，也就是说，我们是在一个具有数千甚至数百万个维度的空间中寻找全局最小值。而且，由于在每次迭代中都要更新权重，并传递不同的 X 和 y，因此**最小值的曲线是不断变化的**！这是多年来神经网络遭到质疑的主要原因之一，似乎用这种方式迭代更新权重并不能真正找到全局最优的解决方案。Yann LeCun 等人在 2015 年《自然》杂志的一篇文章中说得最好：

> 人们普遍认为，简单的梯度下降会陷入不良的局部最小值——对权重配置进行细微调整不会减少平均误差。在实践中，大型神经网络很少会出现不良局部最小值的问题。不管初始条件如何，该系统几乎总是能获得质量非常接近的解决方案。最近的理论和经验结果强烈表明，局部最小值通常不是一个严重的问题。

在实践中，图 4-3 既提供了一个很好的思维模型，说明了学习率不应过大或过小，也足以解释本章将介绍的众多技术是行之有效的。从直觉上理解了神经网络的目标之后，便可以开始研究这些技术。下面将从 **softmax 交叉熵**（softmax cross entropy）损失函数开始研究，该函数之所以起作用，在很大程度上是因为相比第 3 章中的均方误差损失函数，它能够提供更为陡峭的权重梯度。

4.2 softmax交叉熵损失函数

第 3 章使用 MSE（均方误差）作为损失函数，这个函数具有很好的凸性，这意味着预测值距离目标值越远，`Loss` 发送回神经网络 `Layer` 的初始梯度越陡峭，参数接收到的所有梯度也就越大。事实证明，在分类问题中，该函数还可以做得更好，这是因为在该类问题中，**神经网络的输出值应解释为概率**。因此，对于输入到神经网络的所有观测值，不仅每个值都应该位于 0 和 1 之间，而且概率向量的总和应该为 1。softmax 交叉熵损失函数正是利用这一点，来产生比相同输入对应的均方误差损失更为陡峭的梯度。这个函数有两个组件，分别是 softmax 函数和交叉熵损失。接下来依次介绍这两个组件。

4.2.1 组件1：softmax函数

对于具有 N 个可能类别的分类问题，可以让神经网络为每个观测值输出一个包含 N 个值的向量。以具有 3 个类别的问题为例，这些值可以是下面这样。

```
[5, 3, 2]
```

1. 数学

针对分类问题，我们知道应该将输出解释为概率向量。将这些值转换为概率向量的一种方法是将它们归一化，即每个值除以所有值之和：

$$\text{normalize}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{x_1}{x_1 + x_2 + x_3} \\ \frac{x_2}{x_1 + x_2 + x_3} \\ \frac{x_3}{x_1 + x_2 + x_3} \end{bmatrix}$$

然而，有另一种方法，它既能产生更陡峭的梯度，又具有一些优雅的数学属性，那就是使用 softmax 函数。对于长度为 3 的向量，该函数定义如下。

$$\text{softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}} \\ \frac{e^{x_2}}{e^{x_1} + e^{x_2} + e^{x_3}} \\ \frac{e^{x_3}}{e^{x_1} + e^{x_2} + e^{x_3}} \end{bmatrix}$$

2. 理解

在 softmax 函数中，相对于其他值，它可以最大限度地放大最大值，并且在分类问题中，迫使神经网络“不中立”而使其偏向它认为正确的预测。下面比较 normalize 和 softmax 这

两个函数对前面提到的概率向量的作用：

```
normalize(np.array([5,3,2]))
array([0.5, 0.3, 0.2])
softmax(np.array([5,3,2]))
array([0.84, 0.11, 0.04])
```

可以看到，softmax 函数使原始最大值 5 对应的输出值高于 normalize 函数得到的值，而其他两个值则低于它们从 normalize 函数中得出的值。由于 softmax 函数介于将值进行归一化和实际应用 max 函数之间[3]，因此将它命名为 softmax。

4.2.2 组件2：交叉熵损失

回想一下，任何损失函数都包含一个概率向量$\begin{bmatrix} \boldsymbol{p}_1 \\ \vdots \\ \boldsymbol{p}_n \end{bmatrix}$和一个实际值向量$\begin{bmatrix} \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_n \end{bmatrix}$。

1. 数学

对于这些向量中的每个索引值 i，交叉熵损失函数表示如下。

$$\text{CE}(\boldsymbol{p}_i, \boldsymbol{y}_i) = -\boldsymbol{y}_i \times \log(\boldsymbol{p}_i) - (1 - \boldsymbol{y}_i) \times \log(1 - \boldsymbol{p}_i)$$

2. 理解

对于交叉熵损失函数，可以这样思考：因为 y 的每个元素都是 0 或 1，所以前面的方程可以简化为：

$$\text{CE}(\boldsymbol{p}_i, \boldsymbol{y}_i) = \begin{cases} -\log(1 - \boldsymbol{p}_i), & \text{当}\ \boldsymbol{y}_i = 0\ \text{时} \\ -\log(\boldsymbol{p}_i) \quad, & \text{当}\ \boldsymbol{y}_i = 1\ \text{时} \end{cases}$$

现在，可以更轻松地进行分解。如果 $y = 0$，则在 0 到 1 的区间内，该损失值与均方误差损失值的关系如图 4-4 所示。

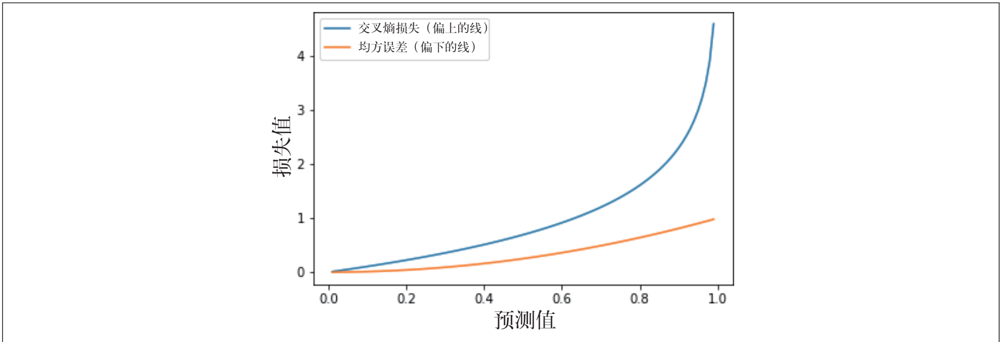


图 4-4：当 $y = 0$ 时，交叉熵损失与均方误差的关系

注 3：在本例中，这将导致输出 `array([1.0, 0.0, 0.0])`。

在这个区间内，不仅对交叉熵损失的惩罚要高得多[4]，而且它们以更快的速率变陡。实际上，当预测值与目标值之差接近 1 时，交叉熵损失的值就接近无穷大！当 $y = 1$ 时的曲线也类似，只是做了“翻转”，即绕 $x = 0.5$ 旋转了 180 度。

因此，对于所知的输出将位于 0 和 1 之间的问题，交叉熵损失产生的梯度比均方误差的梯度更陡。当把这个损失和 softmax 函数结合起来时，真正的“魔术”就出现了：首先通过 softmax 函数将神经网络的输出进行归一化，这样值的总和变为 1；然后将得到的概率输入给交叉熵损失函数。

看一下目前的 3 类场景。从 $i = 1$ 开始的损失向量的组件表达式如下所示（给定观测值的损失的第一个组件，将其表示为 SCE_1）：

$$\mathrm{SCE}_1 = -\boldsymbol{y}_1 \times \log\left(\frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}\right) - (1 - y_1) \times \log\left(1 - \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}\right)$$

基于这个表达式，对于这种损失，梯度似乎要复杂一些。然而，还是存在一个简洁的表达式，既易于数学表达又易于实现：

$$\frac{\partial \mathrm{SCE}_1}{\partial x_1} = \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}} - y_1$$

这意味着 softmax 交叉熵的总梯度为：

$$\mathrm{softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

就是这样！正如所承诺的，最终的实现也很简单：

```
softmax_x = softmax(x, axis = 1)
loss_grad = softmax_x - y
```

接下来对此进行编码。

3. 代码

回顾第 3 章，所有 `Loss` 类都将接收到两个二维数组，一个包含神经网络的预测值，另一个包含目标值。每个数组中的行数是批次大小，列数是分类问题中的类别数 n。每行代表数据集中的一个观测值，该行中的 n 个值代表神经网络对该观测值的最佳猜测，对应所属 n 个类别中每一个类别的概率。因此，必须将 softmax 函数应用于 `prediction` 数组中的**每一行**。这导致了一个潜在的问题：接下来将把得到的数值输入至 log 函数来计算损失。但

注 4：可以更具体地讲：在 0 到 1 的区间内，$-\log(1-x)$ 的平均值为 1，而在相同区间内的 x^2 的平均值仅为 $\frac{1}{3}$。

这样会引发问题，因为当 x 无限接近 0 时，$\log x$ 将变为负无穷大，同样，当 x 无限接近 1 时，$\log(1-x)$ 将变为负无穷大。为防止可能导致数值不稳定的极大损失值，这里限制 softmax 函数的输出取值范围为 10^{-7} ～ 10^{7}。

最后，可以将所有内容整合在一起！

```
class SoftmaxCrossEntropyLoss(Loss):
    def __init__(self, eps: float=1e-9)
        super().__init__()
        self.eps = eps
        self.single_output = False

    def _output(self) -> float:

        # 对每一行（观测值）应用softmax函数
        softmax_preds = softmax(self.prediction, axis=1)

        # 为防止数值不稳定，限制softmax函数的输出的取值范围
        self.softmax_preds = np.clip(softmax_preds, self.eps, 1 - self.eps)

        # 实际损失计算
        softmax_cross_entropy_loss = (
            -1.0 * self.target * np.log(self.softmax_preds) - \
                (1.0 - self.target) * np.log(1 - self.softmax_preds)
        )

        return np.sum(softmax_cross_entropy_loss)

    def _input_grad(self) -> ndarray:

        return self.softmax_preds - self.target
```

很快，本章将通过 MNIST 数据集上的一些实验来说明这种损失如何改善均方误差损失。不过，首先看一下选择激活函数所涉及的考虑因素，查看是否有比 sigmoid 函数更好的选择。

4.2.3 关于激活函数的注意事项

在第 2 章中，我们认为 sigmoid 函数是很好的激活函数，这是因为它具有以下两个特征。

1. 它是非线性单调函数。
2. 它在模型上提供了“正则化”效果，将中间特征强制界定在有限范围内，具体而言就是在 0 和 1 之间。

尽管如此，类似于均方误差损失，sigmoid 函数也有一个缺点：它在后向传递时会**产生相对平坦的梯度**。后向传递到 sigmoid 函数（或任何其他函数）的梯度代表函数的**输出**最终对损失的影响程度。因为 sigmoid 函数的最大斜率是 0.25，所以当把这些梯度向后发送至模型中的上一个运算时，这些梯度**最多**只能除以 4。更糟糕的是，当 sigmoid 函数的输入

小于 –2 或大于 2 时，这些输入所接收的梯度将几乎为 0，这是因为 sigmoid(x) 在 x = –2 或 x = 2 时几乎是平坦的。这意味着影响这些输入的任何参数都会收到较小的梯度，因此神经网络的学习速度会很慢[5]。此外，如果在神经网络的连续层中使用多个 sigmoid 激活函数，那么这个问题将变得更加复杂，进一步减少了神经网络中较早存在的权重可能接收到的梯度。

“处于另一极端”（具有相反的优缺点）的激活函数会是什么情况呢？

1. 另一个极端：ReLU

ReLU[6] 是一种常用的激活函数，它具有与 sigmoid 函数相反的优缺点。如果 x 小于 0，则 ReLU 简单地定义为 0，否则定义为 x，如图 4-5 所示。

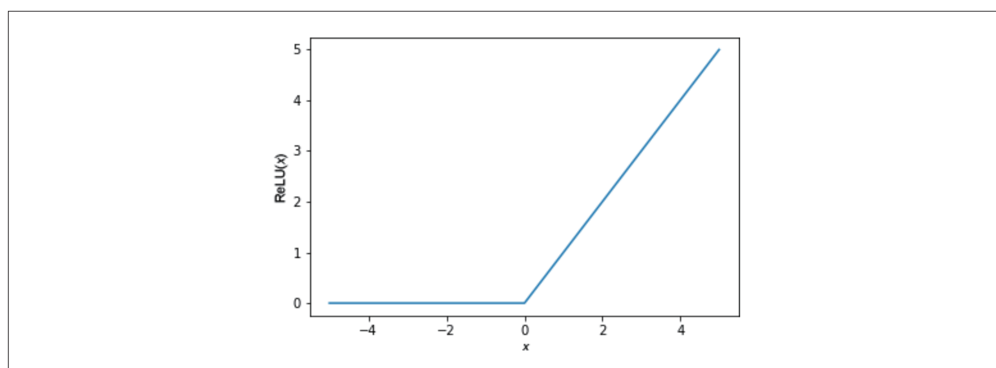


图 4-5：ReLU 激活函数

从单调和非线性的角度来看，这是一个“有效”的激活函数。它产生的梯度要比 sigmoid 函数大得多，即如果函数的输入大于 0，则梯度为 1，其他情况则为 0，平均梯度为 0.5，而 sigmoid 函数可生成的最大梯度为 0.25。ReLU 激活函数在深度神经网络架构中是非常流行的选择，这是因为它的缺点（在小于 0 和大于 0 的值之间形成了过于明显的差别）可以通过其他技术来弥补，包括本章讨论的一些技术，而它的优点（产生大的梯度）对于训练深层神经网络架构中的权重至关重要。

不过，有一个激活函数介于这两者之间，是一种令人愉快的折中方案。本章使用它来演示，它就是 Tanh 函数。

2. 令人愉快的折中方案：Tanh

Tanh 函数的形状与 sigmoid 函数类似，但输出的取值范围是 –1 ～ 1，如图 4-6 所示。

注 5：可以直观了解发生这种情况的原因：假设权重 w 构成了特征 f，即 $f = w \times x_1 + \cdots$，在神经网络的前向传递过程中，当 f = –10 时记录一些观测值。由于 sigmoid(x) 在 x = –10 时近似平坦，因此更改 w 的值几乎不会影响模型预测，也就不会影响损失。

注 6：Rectified Linear Unit，修正线性单元。

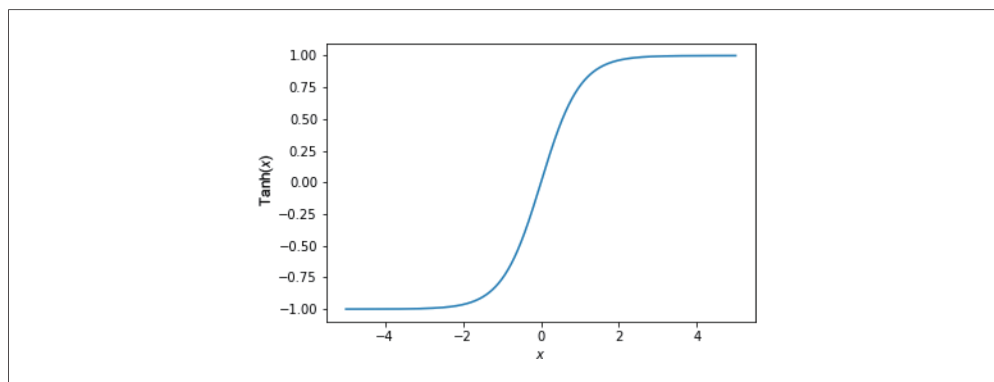


图 4-6：Tanh 激活函数

该函数产生比 sigmoid 曲线更陡峭的梯度。具体来说，与 sigmoid 函数的 0.25 相比，Tanh 函数的最大梯度为 1。图 4-7 展示了这两个函数的导数变化情况。

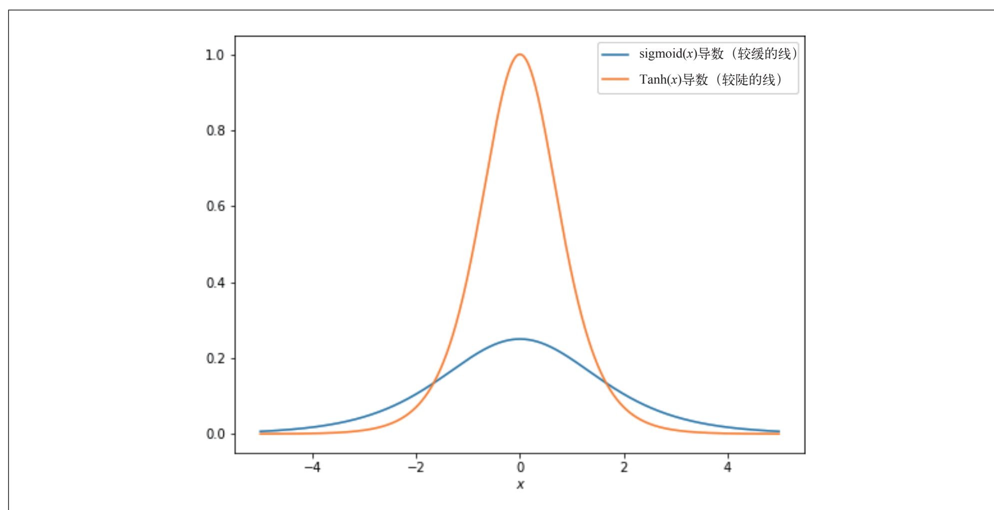


图 4-7：对比 sigmoid 函数与 Tanh 函数的导数

此外，正如 $f(x)=\text{sigmoid}(x)$ 具有易于表达的导数 $f'(x)=\text{sigmoid}(x)\times(1-\text{sigmoid}(x))$ 一样，$f(x)=\tanh(x)$ 也具有易于表达的导数 , 即 $f'(x)=1-\tanh^2(x)$ 。

这里的要点是，无论采用哪种架构，选择激活函数都需要进行权衡：我们想要这样一个激活函数，该函数将使神经网络能够学习输入和输出之间的非线性关系，而又不会增加不必要的复杂性，因为这种复杂性会导致神经网络更难找到一个好的解决方案。例如，当 ReLU 激活函数的输入小于 0 时，Leaky ReLU[7] 激活函数允许略微的负斜率，从而增强

注 7：即“带泄漏的修正线性单元”。——译者注

ReLU 激活函数向后发送梯度的能力；而 ReLU6 激活函数将 ReLU 激活函数的正端限制为 6，从而将更强的非线性引入神经网络。尽管如此，这两个激活函数都比 ReLU 激活函数更为复杂。如果要解决的问题相对简单，那么那些更复杂的激活函数可能会导致神经网络学习起来更为困难。因此，在本书其余部分所演示的模型中，我们将仅使用 Tanh 激活函数，该函数很好地平衡了这些因素。

现在已经选择了激活函数，接下来使用它做一些实验。

4.3 实验

为什么 softmax 交叉熵损失在整个深度学习中如此普遍呢[8]？这里将使用 MNIST 数据集，它由黑白图像组成，每幅图像包含 28 像素 ×28 像素的手写数字，每像素的取值范围为 0（白色）～ 255（黑色）。此外，该数据集被预先分为包含 60 000 幅图像的训练集和 10 000 幅附加图像的测试集。你可以在本书的随书文件中找到一个辅助函数，该辅助函数使用以下代码在训练集和测试集中读取图像及其对应标签：

```
X_train, y_train, X_test, y_test = mnist.load()
```

我们的目标是训练一个神经网络，从而学习图像所包含的具体内容，也就是具体包含 0 ～ 9 中的哪些数字。

4.3.1 数据预处理

对于分类，必须执行**独热编码**（one-hot encoding），从而将表示标签的向量转换为与预测形状相同的 `ndarray`。具体地说，将标签“0”映射到一个向量，该向量的第一个位置为 1（索引为 0），而其他所有位置都为 0，再将标签“1”映射到第二个位置为 1 的向量（索引为 1），以此类推：

$$[0,\ 2,\ 1] \Rightarrow \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \end{bmatrix}$$

最后，就像在前几章中对现实世界的数据集所做的那样，把数据缩放成均值为 0 和方差为 1，这点很有帮助。但是，由于每个数据点都是一幅图像，因此这里不会将每个**特征**缩放为均值 0 和方差 1，否则会导致相邻像素的值变为不同的量，这样可能会使图像失真！相反，仅对数据集进行全局缩放，即减去总体均值并除以标准偏差。注意，使用训练集中的统计数据来缩放测试集。

注 8：举例来说，TensorFlow 的 MNIST 分类教程使用 `softmax_cross_entropy_with_logits` 函数，PyTorch 的 `nn.CrossEntropyLoss` 则实际上在其内部计算 softmax 函数。

```
X_train, X_test = X_train - np.mean(X_train), X_test - np.mean(X_train)
X_train, X_test = X_train / np.std(X_train), X_test / np.std(X_train)
```

4.3.2 模型

必须定义模型，使每个输入都有 10 个输出，即 10 个类别中的每个类别都对应模型中的一个概率值。这样一来，因为每个输出都是概率，所以最后一层将为模型提供一个 sigmoid 激活函数。关于这些“训练技巧”是否真正增强了模型的学习能力，为了说明这一点，本章将使用一个一致性模型架构，该架构基于双层神经网络，其中隐藏层中的神经元数量接近输入数量（784）和输出数量（10）的几何平均值：$89 \approx \sqrt{784 \times 10}$。

现在来看看第一个实验，将经过简单均方误差损失训练的神经网络与经过 softmax 交叉熵损失训练的神经网络进行比较。你所看到的损失值是按观测值进行显示的，回想一下，平均交叉熵损失的绝对损失值是均方误差损失值的 3 倍。运行下面的代码：

```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Sigmoid())],
            loss = MeanSquaredError(),
seed=20190119)

optimizer = SGD(0.1)

trainer = Trainer(model, optimizer)
trainer.fit(X_train, train_labels, X_test, test_labels,
            epochs = 50,
            eval_every = 10,
            seed=20190119,
            batch_size=60);

calc_accuracy_model(model, X_test)
```

将得到如下结果：

```
Validation loss after 10 epochs is 0.611
Validation loss after 20 epochs is 0.428
Validation loss after 30 epochs is 0.389
Validation loss after 40 epochs is 0.374
Validation loss after 50 epochs is 0.366

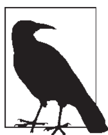
The model validation accuracy is: 72.58%
```

结果显示，模型验证准确度为 72.58%。接下来测试本章稍前得出的结论：softmax 交叉熵损失函数有助于模型更快地学习。

4.3.3 实验：softmax交叉熵损失函数

首先，修改前面的模型。

```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Linear())],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```



由于现在通过 softmax 函数将模型输出作为损失的一部分进行提供，因此不再需要通过 sigmoid 激活函数来提供模型输出。

然后，为模型运行 50 轮，得出以下结果：

```
Validation loss after 10 epochs is 0.630
Validation loss after 20 epochs is 0.574
Validation loss after 30 epochs is 0.549
Validation loss after 40 epochs is 0.546
Loss increased after epoch 50, final loss was 0.546, using the model from epoch 40

The model validation accuracy is: 91.01%
```

模型验证准确度为 91.01%。的确，将损失函数改为提供更陡梯度的函数，仅此一点就可以极大地提高模型的准确性[9]！

当然，不改变架构同样也可以做得更好。4.4 节将讨论动量，这是对迄今为止一直使用的随机梯度下降优化技术最重要且最直接的扩展。

4.4 动量

目前只使用了一个“更新规则”来计算每个时间步长的权重，只需求出损失相对于权重的导数，然后将权重朝正确的方向移动。这意味着 Optimizer 类中的 _update_rule 函数看起来如下所示：

```
update = self.lr*kwargs['grad']
kwargs['param'] -= update
```

先来理解为何要扩展这个更新规则并将动量纳入其中。

注 9：你可能会说，softmax 交叉熵损失在这里获得了“不公平的优势”，因为 softmax 函数将其接收的值归一化为 1。均方误差损失仅获得了 10 个输入，这些输入已经传递给 sigmoid 函数且未归一化为 1。但是，即使将输入归一化为均方误差损失，让它对于每个观测值而言总和都为 1，均方误差损失仍然比 softmax 交叉熵损失要差。

4.4.1 理解动量

回顾图 4-3，它展示了单个参数的值与神经网络的损失值之间的关系。想象这样一种情况：因为每次迭代损失都会不断减少，所以参数的值在相同方向上就会不断更新。这种情况类似于参数“从山上滚下”，并且每个时间步长的更新值都类似于参数的“速度”。但是，在现实世界中，物体不会瞬间停止并改变方向，那是因为它们具有**动量**（momentum），这只是一种简明扼要的说法，即物体在给定瞬间的速度不仅是一个作用在其身上的力的函数，而且是一个基于其过去累积速度的函数（速度越新则权重越大）。这种物理解释是将动量应用到权重更新的动机。接下来详细说明。

4.4.2 在Optimizer类中实现动量

基于动量的参数更新意味着，**每个时间步长的参数更新将是过去时间步长参数更新的加权平均值，其中权重呈指数衰减**。因此，必须选择第二个参数，即动量参数，它将决定这种衰减的程度。动量参数的值越大，每个时间步长的权重更新就越基于参数的累积动量，而不是当前速度。

1. 数学

从数学角度讲，如果动量参数为 μ，并且每个时间步长的梯度为 ∇_t，则权重更新为：

$$\text{update} = \nabla_t + \mu \times \nabla_{t-1} + \mu^2 \times \nabla_{t-2} + \cdots$$

如果动量参数为 0.9，则将 1 个时间步长之前的梯度乘以 0.9，将 2 个时间步长之前的梯度乘以 $0.9^2 = 0.81$，将 3 个时间步长之前的梯度乘以 $0.9^3 = 0.729$，以此类推，最后将所有这些都添加到当前时间步长的梯度中，从而获取当前时间步长的整体权重更新。

2. 代码

如何实现呢？在每次更新权重时都必须计算无穷和吗？

事实证明，还有一种更巧妙的方法。`Optimizer` 类不仅会在每个时间步长收到一个梯度，还会跟踪记录一个代表参数**更新历史**的独立量。然后，我们在每个时间步长上使用当前梯度来更新这个历史记录，并根据该历史记录来计算实际的参数更新。由于动量大致上是基于物理学的类比，因此可以称这个量为“速度”。

如何更新速度？事实证明，可以遵循下面两个步骤。

1. 乘以动量参数。
2. 增加梯度。

结果就是速度在每个时间步长上取以下 3 个值，从 $t = 1$ 开始。

1. ∇_1
2. $\nabla_2 + \mu \times \nabla_1$
3. $\nabla_3 + \mu \times (\nabla_2 + \mu \times \nabla_1) = \nabla_3 + \mu \times \nabla_2 + \mu^2 \times \nabla_1$

至此，就可以将速度用作参数更新！然后，可以将其合并到 Optimizer 类的一个新子类中，这个子类叫作 SGDMomentum 类，它具有 step 函数和 _update_rule 函数，如下所示：

```
def step(self) -> None:
    '''
    如果是第一次迭代，那么初始化每个参数的“速度”。否则，只需应用_update_rule函数。
    '''
    if self.first:
        # 现在将在第一次迭代中设置速度
        self.velocities = [np.zeros_like(param)
                           for param in self.net.params()]
            self.first = False

        for (param, param_grad, velocity) in zip(self.net.params(),
                                                  self.net.param_grads(),
                                                  self.velocities):
            # 将速度输入到_update_rule函数中
            self._update_rule(param=param,
                              grad=param_grad,
                              velocity=velocity)

    def _update_rule(self, **kwargs) -> None:
            '''
            用动量更新SGD的规则。
            '''
            # 更新速度
            kwargs['velocity'] *= self.momentum
            kwargs['velocity'] += self.lr * kwargs['grad']

            # 使用这个更新参数
            kwargs['param'] -= kwargs['velocity']
```

接下来看一下这个新的优化器能否改进神经网络的训练效果。

4.4.3 实验：带有动量的随机梯度下降

现在来基于 MNIST 数据集训练带有隐藏层的神经网络，这里将 optimizer = SGD(lr = 0.1) 替换为 optimizer = SGDMomentum(lr = 0.1, momentum = 0.9)，其余不做任何改变：

```
Validation loss after 10 epochs is 0.441
Validation loss after 20 epochs is 0.351
Validation loss after 30 epochs is 0.345
Validation loss after 40 epochs is 0.338
Loss increased after epoch 50, final loss was 0.338, using the model from epoch 40

The model validation accuracy is: 95.51%
```

可以看到，损失明显较低，而准确度明显较高，这仅仅是因为在参数更新规则中增加了动量[10]！

当然还可以修改学习率，这样也能够在每次迭代中修改参数更新。虽然可以手动更改初始学习率，但也可以使用一些规则在训练期间自动降低学习率。接下来将介绍最常见的规则。

4.5 学习率衰减

> [学习率] 通常是最重要的超参数，必须确保已对其进行优化。
>
> ——Yoshua Bengio，“Practical Recommendations for Gradient-Based Training of Deep Architectures”，2012 年

再次回顾图 4-3，可以看到，随着训练的进行，越来越需要降低学习率。虽然在训练开始时想“加大步长”，但随着不断迭代更新权重，最终将到达一个点，从这个点开始会“跳过”最小值。注意，这不一定会成为问题，因为当接近最小值时，权重与损失之间的关系会“平滑下降”（如图 4-3 所示），在这种情况下，梯度的大小会随着斜率的减小而自动减小。尽管如此，这种情况可能不会发生，即使发生了，学习率衰减也有助于对该过程进行更细粒度的控制。

4.5.1 学习率衰减的类型

学习率衰减有多种方法。最简单的是线性衰减，其中学习率从其初始值线性下降到某个最终值，而实际的衰减则在每轮结束时实现。更准确地说，在时间步长 t 中，如果想要的初始学习率是 α_{start}，而最终学习率是 α_{end}，那么每个时间步长的学习率将是：

$$\alpha_t = \alpha_{\text{start}} - (\alpha_{\text{start}} - \alpha_{\text{end}}) \times \frac{t}{N}$$

其中，N 是总轮数。

另一种同样有效的简单方法是指数衰减法，在这种方法中，每一轮的学习率都以**恒定比例**下降。公式如下：

$$\alpha_t = \alpha \times \delta^t$$

其中，δ 满足以下公式：

注 10：在使用当前批次数据中的信息（除了梯度）来更新参数时，动量只是其中一种方式。另外，附录简要介绍了其他更新规则，这些更新规则也可以在本书提供的 Lincoln 库中浏览。

$$\delta = \frac{\alpha_{\text{end}}}{\alpha_{\text{start}}}^{\frac{1}{N-1}}$$

以上公式实现起来很简单。首先，初始化 `Optimizer` 类，使其具有“最终学习率”`final_lr`，而初始学习率将在整个训练过程中逐渐衰减：

```
def __init__(self,
             lr: float = 0.01,
             final_lr: float = 0,
             decay_type: str = 'exponential')
    self.lr = lr
    self.final_lr = final_lr
    self.decay_type = decay_type
```

然后，在训练开始时，可以调用 `_setup_decay` 衰减函数，该函数计算每轮中学习率的衰减量：

```
self.optim._setup_decay()
```

这些计算将实现刚刚看到的线性和指数学习率衰减公式：

```
def _setup_decay(self) -> None:

    if not self.decay_type:
        return
    elif self.decay_type == 'exponential':
        self.decay_per_epoch = np.power(self.final_lr / self.lr,
                                        1.0 / (self.max_epochs-1))
    elif self.decay_type == 'linear':
        self.decay_per_epoch = (self.lr - self.final_lr) / (self.max_epochs-1)
```

接着，在每轮结束时，实际上都会降低学习率：

```
def _decay_lr(self) -> None:

    if not self.decay_type:
        return

    if self.decay_type == 'exponential':
        self.lr *= self.decay_per_epoch

    elif self.decay_type == 'linear':
        self.lr -= self.decay_per_epoch
```

最后，在每轮的结尾，在 `fit` 函数中从 `Trainer` 调用 `_decay_lr` 函数：

```
if self.optim.final_lr:
    self.optim._decay_lr()
```

现在运行一些实验，检查这些操作能否改善训练效果。

4.5.2 实验：学习率衰减

接下来，尝试使用学习率衰减来训练相同的模型。这里对学习率进行初始化，让运行中的“平均学习率”等于原有学习率（0.1）：对于线性学习率衰减，将学习率初始化为 0.15，然后将其衰减到 0.05；对于指数衰减，将学习率初始化为 0.2，然后将其衰减到 0.05。通过以下代码执行线性衰减：

```
optimizer = SGDMomentum(0.15, momentum=0.9, final_lr=0.05, decay_type='linear')
```

结果如下所示：

```
Validation loss after 10 epochs is 0.403
Validation loss after 20 epochs is 0.343
Validation loss after 30 epochs is 0.282
Loss increased after epoch 40, final loss was 0.282, using the model from epoch 30

The model validation accuracy is: 95.91%
```

通过以下代码执行指数衰减：

```
optimizer = SGDMomentum(0.2, momentum=0.9, final_lr=0.05, decay_type='exponential')
```

结果如下所示：

```
Validation loss after 10 epochs is 0.461
Validation loss after 20 epochs is 0.323
Validation loss after 30 epochs is 0.284
Loss increased after epoch 40, final loss was 0.284, using the model from epoch 30

The model validation accuracy is: 96.06%
```

在以上实验中，“最佳模型”的损失值分别为 0.282 和 0.284，大大低于之前的 0.338！

4.6 节将讨论如何更智能地初始化模型的权重及其背后的原因。

4.6 权重初始化

当输入值为 0 时，sigmoid 和 Tanh 等激活函数的斜率最大，随着输入值远离 0，这些激活函数的曲线会迅速变得平坦。这可能会在一定程度上限制它们的有效性，因为如果许多输入值都与 0 相距较远，那么附加在这些输入上的权重将在后向传递时接收很小的梯度。

事实证明，这是神经网络中的一个主要问题。可以参考 MNIST 神经网络中的隐藏层，该层将接收 784 个输入，然后将它们乘以权重矩阵，最后得到一定数量的神经元（n 个神经元），接着有选择地向每个神经元都添加一个偏差项。图 4-8 展示了在神经网络隐藏层中的这 n 个值输入到 Tanh 激活函数前后的分布。

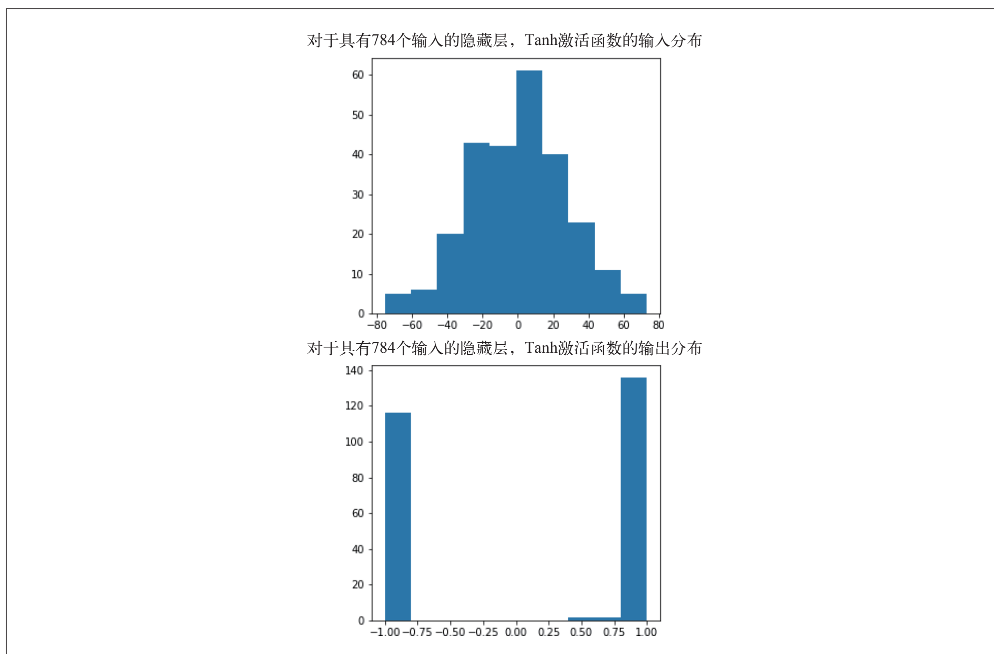


图 4-8：激活函数的输入分布和输出分布

在输入到激活函数之后，大多数激活值是 –1 或 1！这是因为每个特征在数学上都定义为：

$$f_n = w_{1,n} \times x_1 + \cdots + w_{784,n} \times x_{784} + b_n$$

这里初始化了每个权重，让其具有方差 1（$\mathrm{Var}(w_{i,j}) = 1$ 且 $\mathrm{Var}(b_n) = 1$），并且对于独立随机变量 x_1 和 x_2，从 $\mathrm{Var}(x_1 + x_2) = \mathrm{Var}(x_1) + \mathrm{Var}(x_2)$，可以得到：

$$\mathrm{Var}(f_n) = 785$$

它的标准差（$\sqrt{785}$）刚好超过 28，这反映了图 4-8 上半部分显示的分布情况。

这说明上面的操作有问题，但是问题仅仅是提供给激活函数的特征不能“过于分散”吗？如果这是问题所在，那么可以简单地将特征除以某个值来减少其差异。然而，这引出了一个显而易见的问题：如何知道将除以什么值？答案是，这些值**应该根据输入到该层的神经元数量**来进行缩放。如果有一个多层神经网络，一层有 200 个神经元，下一层有 100 个神经元，那么相比具有 100 个神经元的层，具有 200 个神经元的层将传递的值的分布会更广。这是不可取的，我们不希望神经网络在训练中学习的特征的**规模**取决于所传递的特征的数量，同样也不想让神经网络的预测依赖于输入特征的规模。如果将特征中的所有值乘以 2 或除以 2，那么模型的预测结果不应该受到影响。

有几种方法可以修正这个问题。下面将讨论最常用的一种方法：**可以根据连接层中的神经**

元数量调整权重的初始方差，这样在前向传递过程中传递给下一层的值和在后向传递过程中传递给上一层的值具有大致相同的比例。同理，我们必须考虑后向传递，这是因为存在同样的问题：在反向传播期间，因为这是将梯度向后发送的层，所以该层接收的梯度的方差将直接取决于**下一层**中的特征数量。

4.6.1 数学和代码

如果每一层都有 n_{in} 个传入神经元并有 n_{out} 个传出神经元，那么在前向传递过程中，每一个能保持结果特征方差不变的权重的方差将是：

$$\frac{1}{n_{in}}$$

同理，在后向传递过程中，使特征方差保持不变的权重方差为：

$$\frac{1}{n_{out}}$$

作为两者之间的折中方案，通常所说的 **Glorot 初始化**[11]（Glorot initialization）是将每层中的权重方差初始化为：

$$\frac{2}{n_{in}+n_{out}}$$

对其进行编码很简单，在每层中添加 `weight_init` 参数，并在 `_setup_layer` 函数中添加以下内容：

```
if self.weight_init == "glorot":
    scale = 2/(num_in + self.neurons)
else:
    scale = 1.0
```

现在，模型如下所示，其中为每层指定 `weight_init="glorot"`。

```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh(),
                  weight_init="glorot"),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

注 11：这个概念是 Xavier Glorot 和 Yoshua Bengio 在 2010 年发表的论文“Understanding the Difficulty of Training Deep Feedforward Neural Networks”中提出的，因此得名。

4.6.2 实验：权重初始化

运行 4.5.2 节中的模型，但使用 Glorot 初始化来初始化权重，对于具有线性学习率衰减的模型，可以得到如下结果：

```
Validation loss after 10 epochs is 0.352
Validation loss after 20 epochs is 0.280
Validation loss after 30 epochs is 0.244
Loss increased after epoch 40, final loss was 0.244, using the model from epoch 30

The model validation accuracy is: 96.71%
```

对于具有指数学习率衰减的模型，可以得到如下结果：

```
Validation loss after 10 epochs is 0.305
Validation loss after 20 epochs is 0.264
Validation loss after 30 epochs is 0.245
Loss increased after epoch 40, final loss was 0.245, using the model from epoch 30

The model validation accuracy is: 96.71%
```

可以看到，损失显著下降，从先前的 0.282 和 0.284 分别下降到 0.244 和 0.245！注意，通过所有这些更改，**并未增加模型的大小或训练时间**，我们只是凭直觉调整了训练过程。

本章还将介绍最后一种技术。你可能已经注意到，**本章使用的所有模型都不是深度学习模型**。相反，它们只是具有一个隐藏层的神经网络。这是因为如果没有 4.7 节将介绍的 dropout 技术，那么在不发生过拟合的前提下，有效训练深度学习模型极具挑战性。

4.7 dropout

本章展示了对神经网络训练程序的一些修改，使其越来越接近全局最小值。你可能已经注意到，目前本书还没有尝试过看似最显而易见的事情，即在神经网络中增加更多的层，或者为每层增加更多的神经元。原因在于，在大多数神经网络架构中，简单地增加更多"火力"会使神经网络**更难**找到通用的解决方案。尽管能够增加神经网络的容量，使其可以对输入和输出之间的更复杂的关系进行建模，但也有可能导致神经网络找到一个与训练数据过拟合的解决方案。**在大多数情况下，dropout 技术**[12] **可以改变神经网络的容量，并降低发生过拟合的可能性。**

4.7.1 定义

dropout 只是简单地在一层中随机选择一定比例的神经元 p，并在每次前向传递训练中将它们设置为 0。这个奇怪的技巧会改变神经网络的容量，但是从经验上讲，它在许多情况下确实可以防止神经网络过拟合。这在更深层的神经网络中尤为突出，其中学习的特征是从

注 12：以下简称 dropout。——编者注

原始特征中移除的多个抽象层。

尽管 dropout 可以帮助神经网络在训练期间避免过拟合，但我们仍然希望在预测时给神经网络一个做出正确预测的“最佳机会”。因此，`Dropout` 类具有两种模式：应用了 dropout 的训练模式和没有应用 dropout 的推理模式。但是，这带来了新的问题：如果后续各层的权重期望值为 M，则它们将得到的值大小为 $M \times (1-p)$ 。因此，当在推理模式下运行神经网络时，如果要模拟这种幅度变化，除了删除 dropout，还需将所有值乘以 $(1-p)$ 。

为了更清楚地说明这一点，接下来对其进行编码。

4.7.2 实现

可以把 dropout 作为一个 `Operation` 类来实现，将其附加到每一层的末尾，如下所示：

```
class Dropout(Operation):

    def __init__(self,
                 keep_prob: float = 0.8):
        super().__init__()
        self.keep_prob = keep_prob

    def _output(self, inference: bool) -> ndarray:
        if inference:
            return self.inputs * self.keep_prob
        else:
            self.mask = np.random.binomial(1, self.keep_prob,
                                           size=self.inputs.shape)
            return self.inputs * self.mask

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad * self.mask
```

在前向传递中应用 dropout 时，需要保存一个“掩码”，来表示被设置为 0 的单个神经元。然后，在后向传递中，将运算接收到的梯度乘以该掩码。这是因为对于被清零的输入值，dropout 使其梯度为 0（因为现在更改它们的值对损失没有影响），而其他梯度保持不变。

通过调整框架来适应 dropout

你可能已经注意到，`_output` 方法包含 `inference` 标志，用于显示其中是否应用了 dropout。为了正确调用此标志，现在必须在整个训练过程中将其添加到其他几个位置。

1. `Layer` 类和 `NeuralNetwork` 类的 `forward` 方法将 `inference` 作为参数（默认情况下为 `False`），并将该标志传递到每个 `Operation` 类中，这样便可以区别每个 `Operation` 类在训练模式下的行为与在推理模式下的行为。
2. 回想一下，在 `Trainer` 中，每隔 `eval_every` 轮就用测试集对已训练的模型进行计算。现在，每当这样做时，都要计算 `inference` 标志为 `True` 的情况。

```
test_preds = self.net.forward(X_test, inference=True)
```

3. 向 `Layer` 类添加 `dropout` 关键字。现在 `Layer` 类的 `__init__` 方法的完整签名如下所示：

```
def __init__(self,
             neurons: int,
             activation: Operation = Linear(),
             dropout: float = 1.0,
             weight_init: str = "standard")
```

然后通过将以下内容添加到该类的 `_setup_layer` 函数中来添加 dropout 运算：

```
if self.dropout < 1.0:
    self.operations.append(Dropout(self.dropout))
```

就是这样！接下来看一下 dropout 的运行情况。

4.7.3　实验：dropout

首先，可以看到，将 dropout 添加到现有模型中确实减少了损失。在第一层添 dropout（值取 0.8），这样做将 20% 的神经元设置为 0，模型如下所示：

```
mnist_soft = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

然后，使用与以前相同的超参数训练模型，即指数权重从初始学习率 0.2 衰减到最终学习率 0.05，结果如下：

```
Validation loss after 10 epochs is 0.285
Validation loss after 20 epochs is 0.232
Validation loss after 30 epochs is 0.199
Validation loss after 40 epochs is 0.196
Loss increased after epoch 50, final loss was 0.196, using the model from epoch 40

The model validation accuracy is: 96.95%
```

与之前看到的相比，损失又一次显著减少：与之前的 0.244 相比，该模型实现了 0.196 的最小损失。

当添加更多层时，dropout 便真正发挥作用了。现在将本章一直使用的模型更改为深度学习模型，定义第一个隐藏层中的神经元数量是之前隐藏层中的两倍（178），第二个隐藏层中的神经元数量则约为之前隐藏层中的二分之一（46）。这样一来，模型如下所示：

```
model = NeuralNetwork(
    layers=[Dense(neurons=178,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=46,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

注意，前两层应用了 dropout。

使用与以前相同的优化器来训练该模型，可以显著降低所获得的最小损失并提高准确度！

```
Validation loss after 10 epochs is 0.321
Validation loss after 20 epochs is 0.268
Validation loss after 30 epochs is 0.248
Validation loss after 40 epochs is 0.222
Validation loss after 50 epochs is 0.217
Validation loss after 60 epochs is 0.194
Validation loss after 70 epochs is 0.191
Validation loss after 80 epochs is 0.190
Validation loss after 90 epochs is 0.182
Loss increased after epoch 100, final loss was 0.182, using the model from epoch 90

The model validation accuracy is: 97.15%
```

重要的是，如果没有 dropout，那么这种改善是不可能实现的。以下是在没有 dropout 的情况下训练相同模型的结果：

```
Validation loss after 10 epochs is 0.375
Validation loss after 20 epochs is 0.305
Validation loss after 30 epochs is 0.262
Validation loss after 40 epochs is 0.246
Loss increased after epoch 50, final loss was 0.246, using the model from epoch 40

The model validation accuracy is: 96.52%
```

在没有 dropout 的情况下，尽管参数的数量和训练时间都超过了原来的两倍，但相比仅具有一个隐藏层的模型，深度学习模型的性能**更差**！这说明了 dropout 对于有效训练深度学习模型的重要性。实际上，dropout 是 2012 年 ImageNet 获奖模型的重要组成部分，该模型开启了现代深度学习时代[13]。可以说，如果没有 dropout，本书就不会出版！

注 13：有关此方面的更多信息，参见 Geoffrey Hinton 等人的论文“Improving Neural Networks by Preventing Co-adaptation of Feature Detectors”。

4.8 小结

本章介绍了一些用于改善神经网络训练效果的常用技术，内容包括如何理解这些技术的工作原理及底层细节。我在此提供一份具有普适性的清单，其中列出了有助于提升神经网络性能的技术。

- 为权重更新规则增加动量，或其他具有类似效果的高级优化技术。
- 使用本章介绍的线性衰减、指数衰减，或者其他更先进的技术（例如余弦衰减），实现随时间衰减学习率。实际上，更有效的学习率计划不仅每轮改变学习率，还根据**测试集的损失**来改变学习率，仅在这种损失无法减少时才降低学习率。可以试着把后者作为一个练习来实现。
- 确保权重初始化的规模与层中的神经元数量有关（这在大多数神经网络库中是默认实现的）。
- 添加 dropout，尤其是神经网络连续包含多个全连接层的情况。

从第 5 章开始，本书将转向讨论专门针对特定领域的高级架构，最先讨论卷积神经网络，它专门用于理解图像数据。加油！

第5章
CNN

本章将介绍 CNN，即**卷积神经网络**（convolutional neural network）。当输入为图像时（神经网络被广泛应用的一种场景），CNN 是用于预测的标准神经网络架构。在第 1 ～ 4 章中，本书仅专注于完全连接的神经网络，将其实现为一系列 `Dense` 层。在介绍本章内容之前，先来回顾一下神经网络的一些关键要素，据此讨论对图像使用不同架构的动机。然后，与介绍其他概念类似，本章将介绍 CNN 的意义：首先从整体上讨论它的工作方式，然后转而讨论底层细节，最后通过从零开始编码卷积运算来详细说明它的工作原理[1]。通过本章的介绍，你可以对 CNN 的工作原理有足够的了解，从而能够使用它来解决问题，并学习 ResNet、DenseNet 以及 Octave 卷积等 CNN 的高级变体。

5.1　神经网络与表征学习

神经网络最初接收观测数据，每个观测值都由 n 个特征进行表示。目前本书已经展示了两个示例，它们分别属于截然不同的领域：第一个是房价数据集，其中每个观测值由 13 个特征组成，每个特征都代表该房屋的数值属性；第二个是 MNIST 手写数字数据集。由于图像由 784 像素（宽为 28 像素、高为 28 像素）表示，因此每个观测值都由 784 来表示单位像素的明暗程度。

在每种情况下，在对数据进行适当缩放后，都可以构建一个模型，从而提供高准确度的预测结果。同样在每种情况下，具有一个隐藏层的简单神经网络模型的性能要优于没有该隐

注 1：本章中的代码虽然能够清楚地说明 CNN 的工作原理，但效率极低。附录在“关于偏差项的损失梯度”中提供了针对批处理、多通道卷积运算的一个更高效的实现，本章将使用 NumPy 库对其进行描述。

藏层的模型。原因是什么？正如房价数据所显示的那样，原因之一是神经网络可以学习输入和输出之间的**非线性关系**。但是，更普遍的原因是，在机器学习中经常需要原始特征的**线性组合**，以便有效地预测目标。假设 MNIST 数字的像素值是从 x_1 到 x_{784}，那么可能出现这样的情况：x_1 大于平均值、x_{139} 小于平均值、x_{237} 也小于平均值，这种组合的预测图像将很可能是数字 9。当然，可能还有许多其他这样的组合，所有这些组合都对图像属于特定数字的概率有正面或负面的影响。在训练过程中，神经网络可以自动发现重要的原始特征组合，该过程首先通过随机权重矩阵乘法产生原始特征的初始随机组合，通过训练，神经网络学习有助于优化的组合而舍弃没有用的组合。这种学习重要特征组合的过程称为**表征学习**（representation learning），这是神经网络在不同领域取得成功的主要原因。图 5-1 有助于理解这个过程。

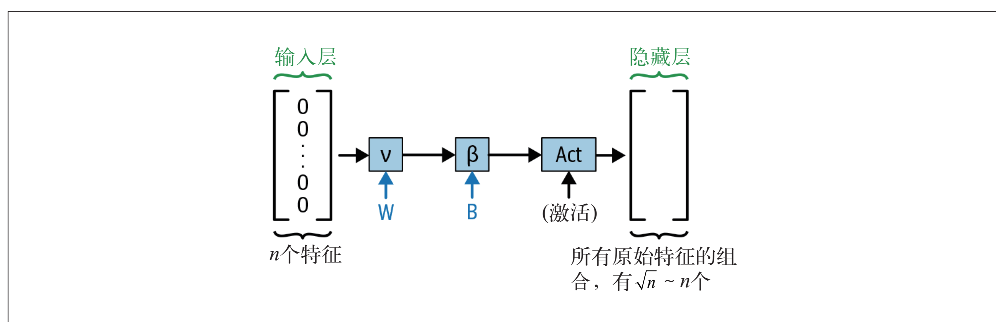


图 5-1：目前介绍的神经网络从 n 个特征开始，然后在 $\sqrt{n}$ 和 n 个特征的组合中进行学习，从而做出预测

对于图像数据，是否有任何理由修改此过程？答案为“是”：**在图像中，有趣的“特征组合”（像素）往往来自图像中彼此靠近的像素**。也就是说，在整幅图像中，相比 3×3 相邻的像素块，9 个随机选择的像素组合更难产生有趣的特征。这里想利用这个关于图像数据的基本事实：特征的顺序很重要，它显示了哪些像素在空间上彼此接近，而在房价数据中，特征的顺序并不重要。

5.1.1 针对图像数据的不同架构

总体而言，解决方案就是向前面那样创建特征组合，但数量级更大，并且每个特征都只来自输入图像中的一个小矩形块像素组合，如图 5-2 所示。

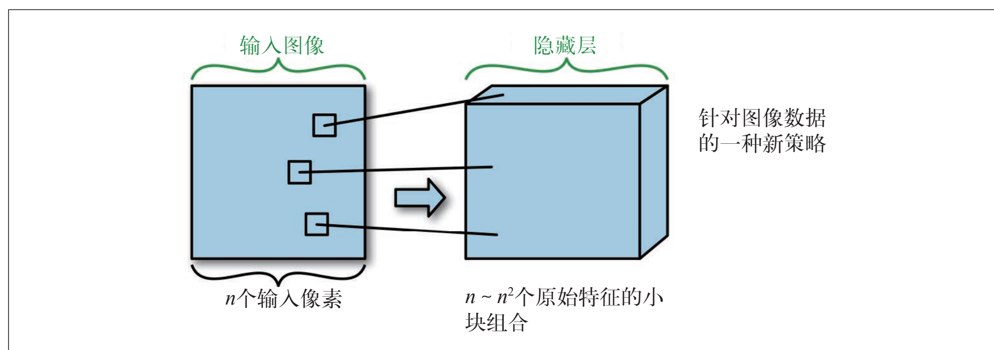


图 5-2：利用图像数据，可以将每个已学习的特征定义为一小块数据的函数，从而定义 n 和 n^2 个输出神经元之间的某个位置

让神经网络学习所有输入特征的组合，即学习输入图像中所有像素的组合，这样做非常低效。原因是，图像的大多数有趣的特征组合出现在小块区域内。不过，至少前文介绍的计算所有输入特征组合的新特征非常容易：如果有 f 个输入特征并且想计算 n 个新特征，那么可以简单地将包含输入特征的 `ndarray` 乘以一个 $f \times n$ 矩阵。要针对输入图像计算局部矩形块中的像素组合，可以采用哪种运算呢？答案就是卷积运算！

5.1.2　卷积运算

在学习卷积运算之前，先要理解这句话的含义：特征是来自图像中一个局部块的像素组合。假设有 3×3 的输入图像 $\boldsymbol{I}$，其满足：

$$\boldsymbol{I} = \begin{bmatrix} i_{11} & i_{12} & i_{13} & i_{14} & i_{15} \\ i_{21} & i_{22} & i_{23} & i_{24} & i_{25} \\ i_{31} & i_{32} & i_{33} & i_{34} & i_{35} \\ i_{41} & i_{42} & i_{43} & i_{44} & i_{45} \\ i_{51} & i_{52} & i_{53} & i_{54} & i_{55} \end{bmatrix}$$

然后，假设要计算一个新特征，该特征是中间 3×3 像素块的函数。就像在神经网络中定义的新特征是旧特征的线性组合一样，这里将定义一个新特征，该新特征是这个 3×3 像素块的函数，通过定义一个 3×3 权重集合 $\boldsymbol{W}$ 来实现：

$$\boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

接着，获取 $\boldsymbol{W}$ 与 $\boldsymbol{I}$ 中相关块的点积，从而得到特征值。由于所涉及的部分以 (3,3) 为中心，因此将其表示为 o_{33}（o 代表“输出”）：

$$o_{33} = w_{11} \times i_{22} + w_{12} \times i_{23} + w_{13} \times i_{24} + w_{21} \times i_{32} + w_{22} \times i_{33} + w_{23} \times i_{34} + w_{31} \times i_{42} + w_{32} \times i_{43} + w_{33} \times i_{44}$$

这样一来，便可以像处理神经网络中的其他计算出的特征一样处理该值：它可能会添加一个偏差项，可能输入给一个激活函数，然后作为“神经元”或“已学习的特征”传递到神经网络中后续的层。因此，我们可以定义特征，这些特征是输入图像中小块像素的函数。

应该如何解释这些特征呢？事实证明，以这种方式计算出的特征具有特殊的含义：表示**由权重定义的视觉模式**是否存在于图像的对应位置。在计算机视觉领域，当用图像中每个位置上的像素值获取其点积时，3×3 或 5×5 之类的数字数组可以表示为“模式检测器”，这一点早已为人们所熟知。举例来说，取以下 3×3 数字数组的点积：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

输入图像的给定部分将检测图像在该位置是否存在边缘。已知有类似的矩阵可以检测是否存在角、是否存在垂直线或水平线等[2]。

现在假设使用相同的权重集 $\boldsymbol{W}$，检测由 $\boldsymbol{W}$ 定义的视觉模式是否存在于输入图像中的每个位置。可以想象“在输入图像上滑动 $\boldsymbol{W}$”，将 $\boldsymbol{W}$ 的点积与图像每个位置的像素相乘，最后得到一幅与原始图像大小几乎相同的新图像 $\boldsymbol{O}$（可能略有不同，具体取决于处理边缘的方式）。该图像 $\boldsymbol{O}$ 将是一种“特征图”，它显示输入图像中由 $\boldsymbol{W}$ 定义的模式所在的位置。实际上，这种运算就是在 CNN 中发生的。我们称之为**卷积运算**（convolution operation），其输出称为**特征图**（feature map）。

卷积运算是 CNN 的核心。前几章介绍了大量有关 `Operation` 类的内容，在向其中引入 CNN 之前，还必须为 CNN 添加另一个维度。

5.1.3　多通道卷积运算

CNN 与常规神经网络的不同之处在于，它创建了更多数量级的特征，并且每个特征都只是一小块输入图像的函数。现在可以得到更具体的结果：从 n 个输入像素开始，刚刚描述的卷积运算将创建 n 个输出特征，每个特征对应输入图像中的一个位置。在神经网络的卷积 `Layer` 类中，执行的操作又更深了一层：创建 f 个集合，每个集合包含 n 个特征，每个特征具有一组对应的（初始随机）权重集，这些权重定义了一种视觉模式，其在输入图像中的每个位置上的检测将被捕获到特征图中。这里的 f 个特征图将通过 f 个卷积运算来创建，如图 5-3 所示。

注 2：可以在 Wikipedia 上搜索“Kernel (image processing)”，浏览更多示例。

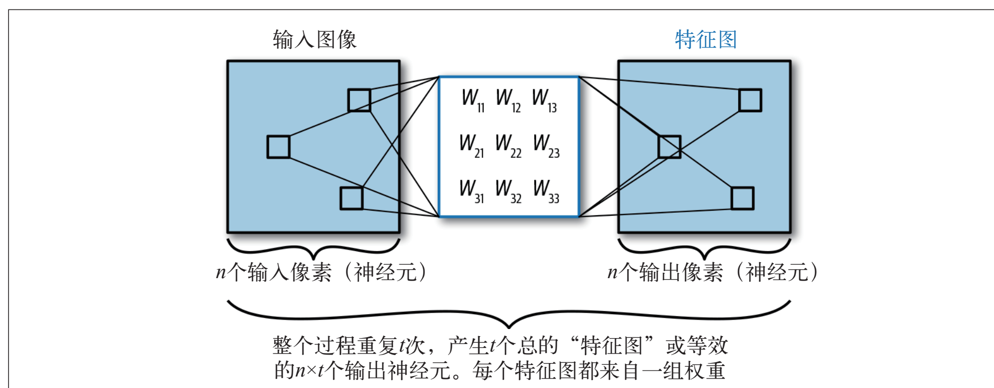


图 5-3：对于具有 n 像素的输入图像，我们定义了具有 f 个特征图的输出，每个特征图的大小与原始图像大致相同，因此该图像总共有 $n \times f$ 个输出神经元，每一个仅是原始图像的一小块像素的函数

本节介绍了许多概念，现在来明确一下它们的定义。虽然由一组特定的权重检测到的每个“特征集”都称为特征图，但在卷积 `Layer` 类的上下文中，特征图的数量叫作该 `Layer` 类的通道（channel）数，因此与该 `Layer` 类相关的运算称作**多通道卷积运算**。另外，f 组权重 W_i 叫作**卷积过滤器**（convolutional filter）或者**内核**（kernel）。

5.2 卷积层

了解多通道卷积运算之后，可以考虑如何将该运算合并到神经网络层中。以前，神经网络层相对简单：它们接收二维 `ndarray` 作为输入，并生成二维 `ndarray` 作为输出。但是，根据前文的描述，卷积层将具有一个三维 `ndarray` 作为**单幅图像**的输出，这 3 个维度包括**通道数**[3] × **图像高度** × **图像宽度**。

这就引申出了一个问题：如何将这个 `ndarray` 向前传递到另一个卷积层，从而创建一个“深度卷积”神经网络？前文已经介绍了如何在具有单个通道和过滤器的图像上执行卷积运算，那么就像将两个卷积层串在一起时所执行的运算，如何才能在具有**多个通道的输入**上执行多通道卷积运算呢？搞清这个问题是理解深度卷积神经网络的关键。

考虑具有全连接层的神经网络。在第一个隐藏层中，假设 h_1 个特征是输入层中所有原始特征的组合。在随后的层中，由于特征是上一层的所有特征的组合，因此可能有 h_2 个特征，相当于原始特征的“特征的特征”。为了创建下一层的 h_2 个特征，使用 $h_1 \times h_2$ 个权重来表示每组的 h_2 个特征是上一层中每组 h_1 个特征的函数。

如 5.1 节所述，在 CNN 的第一层中发生了这样一个过程：使用 m_1 个**卷积过滤器**将输入图像转换为 m_1 个**特征图**。该层的输出应该能够说明，在输入图像的每个位置处是否存在由

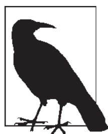
注 3：与“特征图”相同。

m_1 个过滤器的权重表示的 m_1 个视觉模式。正如全连接神经网络的不同层可以包含不同数量的神经元一样，CNN 的下一层可以包含 m_2 个过滤器。为了使神经网络能够学习复杂的模式，每个模式的解释应当是：在图像的该位置处，是否存在**由上一层的 m_1 个视觉模式的组合所表示的每一个“模式的模式”**或高阶视觉特征。这意味着如果卷积层的输出是形状为 m_2 个通道数 × 图像高度 × 图像宽度的三维 `ndarray`，则对于任意 m_2 个特征图，其中给定位置的图像是一个线性组合，**该线性组合会将 m_1 个过滤器卷积到上一层的对应 m_1 个特征图中的所有相同位置**。这样一来，对于任意 m_2 个过滤器图，其中的每个位置都可以表示为先前卷积层中已经学习的 m_1 个视觉特征的**组合**。

5.2.1　实现意义

在理解两个多通道卷积层的连接原理后，这个运算便容易理解了：正如需要 $h_1 \times h_2$ 个权重将具有 h_1 个神经元的全连接层连接到具有 h_2 个神经元的层一样，需要 $m_1 \times m_2$ 个**卷积过滤器**将具有 m_1 个通道的卷积层连接到具有 m_2 个通道的层。因此，现在可以指定 `ndarray` 的维度，这些维度将构成完整的多通道卷积运算的输入、输出和参数。

1. 输入将具有以下形状。
    - 批次大小
    - 输入通道
    - 图像高度
    - 图像宽度
2. 输出将具有以下形状。
    - 批次大小
    - 输出通道
    - 图像高度
    - 图像宽度
3. 卷积过滤器本身将具有以下形状。
    - 输入通道
    - 输出通道
    - 过滤器高度
    - 过滤器宽度



每个神经网络库的维度顺序可能不同，但上述 4 个维度始终存在。

在本章稍后实现卷积运算时，请牢记以上要点。

5.2.2 卷积层与全连接层的区别

本章稍前粗略地讨论了卷积层与全连接层的区别。在了解卷积层的更多细节后，再来看看这个对比，如图 5-4 所示。

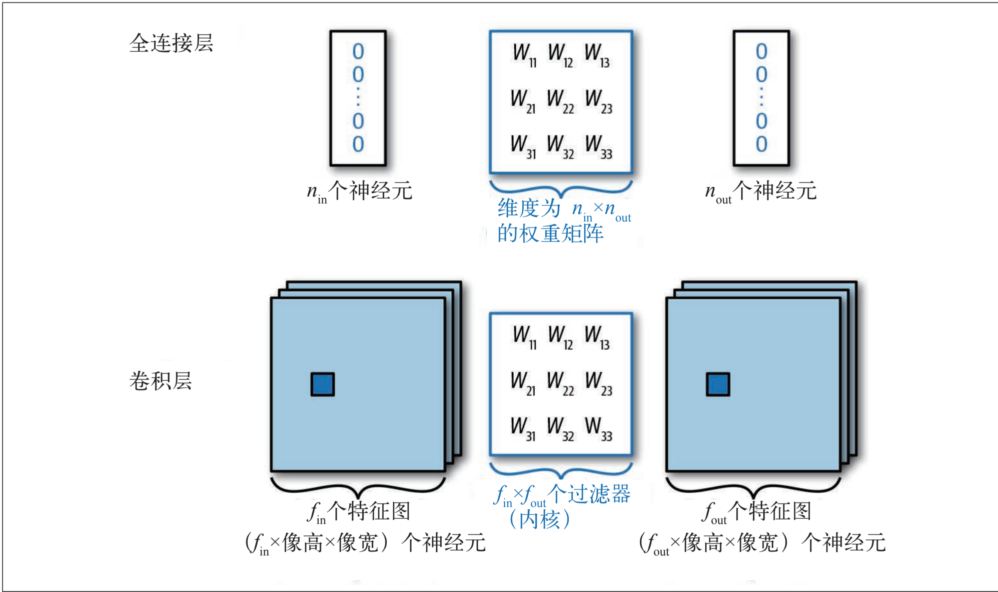


图 5-4：卷积层与全连接层的区别

此外，这两种层还有一个区别，那就是单个神经元本身的解释方式。

- 对全连接层的神经元来说，它检测在当前观测结果中是否存在**由上一层学到的特征的特定组合**。
- 对卷积层的神经元来说，它检测在输入图像的**给定位置处**是否存在由上一层学习的**视觉模式的特定组合**。

在将这样的层合并到神经网络中之前，还需要解决另一个问题：对于获得的多维 `ndarray` 输出，如何使用它们进行预测？

5.2.3 利用卷积层进行预测：`Flatten`层

前文对卷积层进行了大致介绍，包括如何学习表示图像中是否存在视觉模式的特征并将这些特征存储到特征图中，那么如何使用特征图进行预测呢？在第 4 章中，当使用完全连接的神经网络预测图像所属的具体类别时，只需确保最后一层的维数为 10。然后，可以将这 10 个数字输入到 softmax 交叉熵损失函数中，确保它们被处理为概率。现在需要弄清楚在卷积层中执行的运算，在这一层中有一个三维 `ndarray` 对应每次观测，这些观测值的形状为 m 个通道 × 像高 × 像宽。

回想一下，每个神经元仅仅代表在图像中的给定位置处是否存在特定的视觉特征组合。如果这是一个深度卷积神经网络，则可能是特征的特征或特征的“特征的特征”。相比将完全连接的神经网络应用于该图像，在这两种情况下所学到的特征是一样的：第一个全连接层将表示单个像素的特征，第二个全连接层将表示这些特征的特征，以此类推。在全连接架构中，只需将神经网络所学习到的每一个“特征的特征”简单地视为一个神经元，就可以将其用作预测图像所属类别的输入。

事实证明，可以使用 CNN 实现同样的操作。将 m 个特征图视为 $m \times image_{\text{height}} \times image_{\text{width}}$ 个神经元，并使用 `Flatten` 运算将这 3 个维度（通道数、像高、像宽）压缩成一维向量，然后可以使用简单的矩阵乘法进行最终预测。之所以能够这样做，是因为每个神经元在**根本上都代表与全连接层中的神经元相同的“事物”**，具体来说，就是给定的视觉特征或特征组合是否存在于图像中的给定位置处。因此，可以在神经网络的最后一层以同样的方式处理它们[4]。

本章稍后会介绍如何实现 `Flatten` 层。但是，在深入研究实现方法之前，先来讨论另一种层，尽管本书不会详细介绍，但是它在许多 CNN 架构中非常重要。

5.2.4　池化层

在 CNN 中，另一种常用的层是**池化层**（pooling layer）。它对卷积运算创建的每个特征图进行**降采样**（downsample）。对于最常用的大小为 2 的池，在**最大池**的情况下，这涉及将每个特征图的所有 2×2 段映射到该段的最大值；同理，在**平均池**的情况下，则映射到该段的平均值。然后，对于 $n \times n$ 图像，这将把整幅图像映射到大小为 $\frac{n}{2} \times \frac{n}{2}$ 的图像。图 5-5 对此进行了说明。

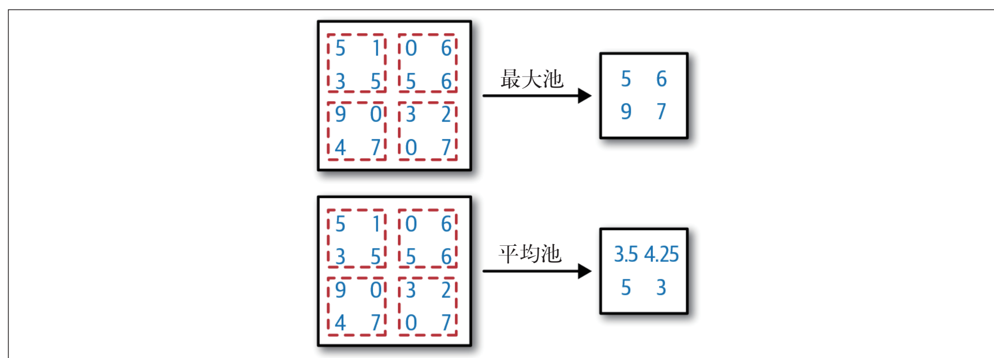


图 5-5：输入为 4 × 4 的最大池和平均池的示意图。每个 2 × 2 块都映射到该像素块的平均值或最大值

注 4：这也说明了理解卷积运算的输出很重要，因为它既要创建多个过滤器图（比方说 m 个），又要创建 $m \times image_{\text{height}} \times image_{\text{width}}$ 个独立神经元。就像整个神经网络一样，关键是要一次在头脑中掌握对多个层次的解释，同时看到它们之间的联系。

池化的主要优点是计算能力：通过对图像进行降采样，使其包含的像素数为上一层的 $\frac{1}{4}$，池化让权重的数量和训练神经网络所需的计算量减少了 $\frac{3}{4}$。如果在神经网络中使用多个池化层（如 CNN 早期在许多架构中使用的池化层），则这可能会进一步得到简化。当然，池化的缺点是只能从降采样的图像中提取 $\frac{1}{4}$ 的信息。然而，尽管使用了池，这种架构在图像识别基准测试中仍表现出非常强的性能，这一事实表明，即使池化通过降低图像分辨率导致神经网络“丢失了图像信息”，但在提高计算速度方面的权衡是值得的。尽管如此，许多人认为池化只是一个碰巧奏效但可能应该被废除的技术。正如 Geoffrey Hinton 于 2014 年所写：“在卷积神经网络中使用池化运算是一个大错误，而它运行得很好这一事实是一场灾难。”实际上，ResNet[5] 等最新的 CNN 架构尽可能不使用池或者根本不使用池。因此，本书不会实现池化层，但是考虑到内容的完整性以及它在推广 CNN 方面所扮演的重要角色（通过在 AlexNet 等著名架构中的应用），本书在此对它进行了介绍。

在图像之外应用 CNN

到目前为止，本书对于使用神经网络处理图像的所有描述都非常标准：图像通常表示为 m_1 个像素通道（$m_1 = 1$ 表示黑白图像，$m_1 = 3$ 表示彩色图像），然后对每个通道应用 m_2 个卷积运算（如前所述，使用 $m_1 \times m_2$ 个过滤器图），并且这种模式会持续进行几层。CNN 的其他处理方法都涵盖了上述内容，但有一点较少提及，那就是将数据组织到通道中，然后使用 CNN 处理这些数据，这不仅仅局限于图像。例如，这种数据表示是 AlphaGo 系列程序的关键，这些程序表明神经网络可以学习下围棋。可以参考以下论文节选内容[6]：

> 神经网络的输入是由 17 个二进制特征平面组成的 $19 \times 19 \times 17$ 图像栈。8 个特征平面 X_t 由二进制值组成，表示当前玩家的棋子。如果交叉点 i 在时间步长 t 包含玩家颜色的棋子，则 $X_t^i = 1$；如果交叉点为空，或者包含对手棋子，又或者 $t < 0$，则 $X_t^i = 0$。另外 8 个特征平面 Y_t，表示对手棋子的相应特征。最后一个特征平面 C 表示要下棋子的颜色，如果要下黑色棋子，则其值为常量 1；如果要下白色棋子，则其值为常量 0。这些平面被连接在一起，从而给出输入特征 $s_t = X_t, Y_t, X_{t-1}, Y_{t-1}, \cdots, X_{t-7}, Y_{t-7}, C$。历史特征“$X_t, Y_t$”很重要，因为这里禁止重复，所以仅从当前的棋子出发对围棋而言是不可观测的。同理，因为围棋的贴目（komi）也是不可观测的，所以颜色特征 C 也很重要。

换句话说，AlphaGo 系列程序本质上将棋盘表示为具有 17 个通道且大小为 19 像素 ×19 像素的“图像”！它们使用其中的 16 个通道对每个玩家之前进行的 8 次移动进行了编码。这是必要做法，这样一来，对于避免连续重复移动的规则，就可以进行编码。第 17 个通

注 5：参见何恺明等人于 2017 年发表的关于 ResNet 的原始论文“Deep Residual Learning for Image Recognition”。

注 6：参见 DeepMind（David Silver 等人）于 2017 年发表的论文“Mastering the Game of Go Without Human Knowledge”。

道实际上是一个 19 × 19 的网格，全为 1 或全为 0，具体取决于要移动的目标[7]。CNN 及其多通道卷积运算通常应用于图像，但更普遍的做法是用多个通道来表示沿某个空间维度排列的数据，这点甚至适用于图像之外的场景。

但是，要真正理解多通道卷积运算，必须从零开始实现它。接下来详细描述这个过程。

5.3 实现多通道卷积运算

事实证明，如果首先考虑一维的情况，那么看似复杂的运算（包含一个四维输入 `ndarray` 和一个四维参数 `ndarray`）实现起来会更加清晰。从这个起点开始构建完整的运算主要是要添加一系列 `for` 循环，整个过程将采用第 1 章介绍的方法，穿插介绍示意图、数学和可运行的 Python 代码。

5.3.1 前向传递

从概念上看，一维卷积与二维卷积相同：将一维输入和一维卷积过滤器作为输入，然后通过沿输入滑动过滤器来创建输出。

$$输入：[t_1, t_2, t_3, t_4, t_5]$$

假设输入长度为 5，并且要检测的“模式”的长度是 3，如下所示。

$$过滤器：[w_1, w_2, w_3]$$

1. 数学

将输入的第一个元素与过滤器进行卷积，来创建输出的第 1 个元素：

$$输出特征\ O_1：t_1w_1 + t_2w_2 + t_3w_3$$

将过滤器向右滑动一个单位并将其与序列中的下一组值进行卷积，可以创建输出的第 2 个元素：

$$输出特征\ O_2：t_2w_1 + t_3w_2 + t_4w_3$$

到目前为止，进展都很顺利。但是，当计算下一个输出值时，可以看到已经没有空间了：

$$输出特征\ O_3：t_3w_1 + t_4w_2 + t_5w_3$$

注 7：DeepMind 使用一个类似于国际象棋的表示法发布了结果。仅有这一次，为了对更复杂的国际象棋规则集进行编码，输入具有 119 个通道！参见 DeepMind（David Silver 等人）于 2018 年发表的论文“A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play”。

现在已经到达了输入的末尾，一开始有 5 个输入，但输出结果只有 3 个元素。如何解决这个问题呢？

2. 填充

为了避免因卷积运算而导致输出缩小，这里将引入一个贯穿 CNN 的技巧：在输入的边缘“填充”零，让输出与输入保持相同的大小。否则，如前所述，每次在输入上卷积过滤器时，都会得到一个比输入稍小的输出。

从前面的卷积示例中，可以得出这样的结论：对于大小为 3 的过滤器，为了让输出与输入的大小相同，边缘周围应该有 1 个填充单元。更具体地说，由于几乎总是使用奇数个过滤器，因此对于填充数量，其大小就是过滤器大小除以 2 后得到的整数部分。

填充之后，输入就不再是从 i_1 到 i_5，而是从 i_0 到 i_6，其中 i_0 和 i_6 都是 0。卷积的输出可以这样计算：

$$o_1 = i_0 \times w_1 + i_1 \times w_2 + i_2 \times w_3$$

以此类推，直到输出的大小与输入的大小相同：

$$o_5 = i_4 \times w_1 + i_5 \times w_2 + i_6 \times w_3$$

那么，该如何编码呢？

3. 代码

这一部分的代码编写起来非常简单。在开始编码之前，先总结要点。

(1) 最终希望生成与输入大小相同的输出。
(2) 要在不“缩小”输出的情况下完成此操作，首先需要填充输入。
(3) 必须编写某种遍历输入的循环，并利用过滤器对每个输入进行卷积。

下面将从输入和过滤器开始：

```
input_1d = np.array([1,2,3,4,5])
param_1d = np.array([1,1,1])
```

以下辅助函数可以在每一端填充一维输入：

```
def _pad_1d(inp: ndarray,
            num: int) -> ndarray:
    z = np.array([0])
    z = np.repeat(z, num)
    return np.concatenate([z, inp, z])

_pad_1d(input_1d, 1)

array([0., 1., 2., 3., 4., 5., 0.])
```

注意，对于要生成的输出中的每个元素，在所填充的输入中都有一个对应的元素，在那里“开始”卷积运算。一旦弄清楚卷积运算的位置，只需循环遍历过滤器中的所有元素，对每个元素执行乘法运算，然后将结果汇总起来。

如何找到这个“对应元素”呢？很简单，注意，第一个输出元素从所填充输入的第一个元素开始获取值！这样一来，for 循环非常容易编写：

```
def conv_1d(inp: ndarray,
            param: ndarray) -> ndarray:

    # 正确断言维度
    assert_dim(inp, 1)
    assert_dim(param, 1)

    # 填充输入
    param_len = param.shape[0]
    param_mid = param_len // 2
    input_pad = _pad_1d(inp, param_mid)

    # 初始化输出
    out = np.zeros(inp.shape)

    # 执行一维卷积
    for o in range(out.shape[0]):
        for p in range(param_len):
            out[o] += param[p] * input_pad[o+p]

    # 确保形状没有改变
    assert_same_shape(inp, out)

    return out

conv_1d_sum(input_1d, param_1d)

array([ 3.,  6.,  9., 12.,  9.])
```

这个过程很简单。在继续进行此运算的后向传递（棘手的部分）之前，简要讨论一下关于卷积的一个被忽略的超参数：步幅。

4. 关于步幅的注意事项

如前所述，池化运算是从特征图中对图像进行降采样的一种方法。在许多早期的卷积架构中，在不对计算准确度造成任何重大影响的情况下，池化确实显著地减少了所需的计算量。不过，它现在不再受欢迎，正是因为它有效地对图像进行了降采样，才导致传送到下一层的图像只有一半分辨率。

还有一种应用更广泛的方法，那就是修改卷积运算的**步幅**（stride），即过滤器在图像上逐渐滑动的量。前面的示例所使用的步幅是 1，即每个过滤器都与输入的每个元素进行卷积，这样输出最终就可以与输入的大小相同。当步幅为 2 时，过滤器将与输入图像的每两个元素进行卷积，这样输出大小将是输入的一半。如果步幅为 3，则过滤器将与输入图像的每

3 个元素进行卷积，以此类推。这意味着，对于使用大小为 2 的池与使用大小为 2 的步幅，它们在计算量方面会减少相同的量，也会产生大小相同的输出，但后者不会损失太多的信息。对于大小为 2 的池，输入中只有 $\frac{1}{4}$ 的元素会影响输出，而当步幅为 2 时，输入中的每个元素都会对输出有一定的影响。因此，即使在当今最先进的 CNN 架构中，使用步幅（大于 1）也比使用池化进行降采样要普遍得多。

不过，本书仅展示步幅为 1 的示例，至于如何修改这些运算，以实现步幅大于 1，你可以自行练习。当然，使用大小为 1 的步幅也使编写后向传递更为容易。

5.3.2　后向传递

在卷积运算中，后向传递比较棘手。回顾在后向传递过程中要执行的操作：5.3.1 节使用输入和参数生成了卷积运算的输出，现在需要进行以下计算。

- 损失相对于卷积运算的每个输入元素的偏导数（之前，输入为 `inp`）。
- 损失相对于每个过滤器元素的偏导数（之前，过滤器为 `param_1d`）。

第 3 章介绍过 `ParamOperation` 类的工作方式。在 `backward` 方法中，`ParamOperation` 类接收一个输出梯度，表示输出的每个元素最终对损失的影响程度，然后使用此输出梯度来计算输入和参数的梯度。因此，需要编写一个函数，该函数接受形状与输入相同的输出梯度（`output_grad`），并生成输入梯度（`input_grad`）和参数梯度（`param_grad`）。

如何测试计算出的梯度是否正确呢？这里将从第 1 章中找到答案：在一个求和公式中，每个元素相对于其输出的偏导数都为 1，即如果 $s = a + b + c$，则有 $\frac{\partial s}{\partial a} = \frac{\partial s}{\partial b} = \frac{\partial s}{\partial c} = 1$。因此，可以使用 `_input_grad` 函数和 `_param_grad` 函数（稍后将介绍并编写）以及 `output_grad`（所有元素都是 1）来计算 `input_grad` 和 `param_grad`。然后以一定的量 α 改变输入元素，检查这些梯度是否正确，并查看得到的和的变化量是否等于梯度乘以 α。

1. 梯度“应该”是多少

现在使用刚才描述的逻辑，来计算输入梯度向量中的元素：

```
def conv_1d_sum(inp: ndarray,
                param: ndarray) -> ndarray:
    out = conv_1d(inp, param)
    return np.sum(out)

# 随机选择将第5个元素加1
input_1d_2 = np.array([1,2,3,4,6])
param_1d = np.array([1,1,1])

print(conv_1d_sum(input_1d, param_1d))
print(conv_1d_sum(input_1d_2, param_1d))

39.0
41.0
```

第 5 个元素的梯度应该是 $41-39=2$ 。

接下来看一下如何计算这样一个梯度，而不是简单地计算两个和之差。

2. 计算一维卷积的梯度

可以看到，增加输入元素会使输出的元素数量加 2。仔细观察输出，可以清楚地理解这个过程：

$$
\begin{aligned}
\text{Output}:[\ & t_0w_1+t_1w_2+t_2w_3, \Rightarrow O_1 \\
& t_1w_1+t_2w_2+t_3w_3, \Rightarrow O_2 \\
& t_2w_1+t_3w_2+t_4w_3, \Rightarrow O_3 \\
& t_3w_1+t_4w_2+t_5w_3, \Rightarrow O_4 \\
& t_4w_1+t_5w_2+t_6w_3] \Rightarrow O_5
\end{aligned}
$$

输入元素 t_5 在输出中出现了两次。

- 作为 O_4 的一部分，t_5 乘以 w_3 。
- 作为 O_5 的一部分，t_5 乘以 w_2 。

注意，如果存在 O_6 ，则 t_5 也将乘以 w_1 ，从而增加输出，这一点有助于了解输入映射到输出总和的通用模式。

因此，t_5 最终影响损失的量将为（可以将其表示为 $\frac{\partial L}{\partial t_5}$ ）：

$$
\frac{\partial L}{\partial t_5}=\frac{\partial L}{\partial O_4}\times w_3+\frac{\partial L}{\partial O_5}\times w_2+\frac{\partial L}{\partial O_6}\times w_1
$$

当然，在这个简单的例子中，当损失仅为总和时，输出中（除了数量为 0 的“填充”元素）所有元素满足 $\frac{\partial L}{\partial O_i}=1$ 。这个总和很容易计算：它就是 w_2+w_3 ，因为 $w_2=w_3=1$ ，所以总和是 2。

3. 通用模式

现在寻找输入元素的通用模式。事实证明，这是跟踪索引的一种练习。由于这里将数学转换为代码，因此使用 o_i^{grad} 表示输出梯度的第 i 个元素（最终将通过 `output_grad[i]` 来访问）。这样便能得到：

$$
\frac{\partial L}{\partial t_5}=o_4^{\text{grad}}\times w_3+o_5^{\text{grad}}\times w_2+o_6^{\text{grad}}\times w_1
$$

仔细观察这个输出，可以得出类似的结论：

$$
\frac{\partial L}{\partial t_3}=o_2^{\text{grad}}\times w_3+o_3^{\text{grad}}\times w_2+o_4^{\text{grad}}\times w_1
$$

$$\frac{\partial L}{\partial t_4} = o_3^{\text{grad}} \times w_3 + o_4^{\text{grad}} \times w_2 + o_5^{\text{grad}} \times w_1$$

显然存在一个模式，但把它转换成代码有点棘手，特别是这里在输出索引增加的同时，权重索引却在减少。尽管如此，仍然可以借助双层 for 循环来实现：

```
# param：示例中形状为(1,3)的ndarray
# param_len：整数3
# inp：示例中形状为(1,5)的ndarray
# input_grad：形状始终与inp相同的ndarray
# output_pad：示例中形状为(1,7)的ndarray
for o in range(inp.shape[0]):
    for p in range(param.shape[0]):
        input_grad[o] += output_pad[o+param_len-p-1] * param[p]
```

这样可以适当增加权重索引，同时减少输出上的权重。

尽管现在可能尚不明显，但是通过计算来理解这一点的确是计算卷积运算梯度最为棘手的部分。如果继续增加复杂性，比如批次大小、具有二维输入的卷积或具有多个通道的输入等，那么只需要在前几行中添加更多的 for 循环，本章稍后会介绍这一点。

4. 计算参数梯度

如果增加过滤器中某元素的值，那么可以用类似的方法来计算输出值的增加情况。将过滤器的第一个元素（或其他元素）增加一个单位，观察其对总和的影响：

```
input_1d = np.array([1,2,3,4,5])
# 随机选择将第1个元素加1
param_1d_2 = np.array([2,1,1])

print(conv_1d_sum(input_1d, param_1d))
print(conv_1d_sum(input_1d, param_1d_2))

39.0
49.0
```

可以看到，$\frac{\partial L}{\partial w_1} = 10$ 。

正如对输入所做的那样，仔细检查输出，查看过滤器中对其产生影响的元素，同时填充输入，这样可以更清楚地观察模式：

$$w_1^{\text{grad}} = t_0 \times o_1^{\text{grad}} + t_1 \times o_2^{\text{grad}} + t_2 \times o_3^{\text{grad}} + t_3 \times o_4^{\text{grad}} + t_4 \times o_5^{\text{grad}}$$

对于总和，由于所有 o_i^{grad} 元素均为 1，且 t_0 为 0，因此可以得到：

$$w_1^{\text{grad}} = t_1 + t_2 + t_3 + t_4 = 1 + 2 + 3 + 4 = 10$$

这证实了前面的计算结果。

5. 编写代码

相比对输入梯度进行编码，这次编码较为容易，因为这一次“索引沿同一方向移动”。在同一个嵌套 for 循环中，代码为下面这种形式：

```
# param：示例中形状为(1,3)的ndarray
# param_grad：和param形状相同的ndarray
# inp：示例中形状为(1,5)的ndarray
# input_pad：形状为(1,7)的ndarray
# output_grad：示例中形状为(1,5)的ndarray
for o in range(inp.shape[0]):
    for p in range(param.shape[0]):
        param_grad[p] += input_pad[o+p] * output_grad[o]
```

最后，可以将这两个计算结合起来，编写一个函数来同时计算输入梯度和过滤器梯度，相关步骤如下。

1. 将输入和过滤器作为参数。
2. 计算输出。
3. 填充输入和输出的梯度，也就是说获取 input_pad 和 output_pad。
4. 如前所示，使用填充的输出梯度和过滤器来计算输入梯度。
5. 同样，使用输出梯度（未填充）和填充的输入来计算过滤器梯度。

本书的随书文件提供了包括以上代码块的完整函数。

以上就是对一维卷积的实现，接下来将这种计算扩展到二维输入、批量二维输入和多通道批量二维输入。不必担心，这些过程都非常简单。

5.3.3 批处理

要处理批量二维输入，先要为这些卷积函数添加功能，其第一维表示输入的批次大小，第二维表示一维序列的长度：

```
input_1d_batch = np.array([[0,1,2,3,4,5,6],
                           [1,2,3,4,5,6,7]])
```

可以遵循之前定义的步骤：首先填充输入，使用它来计算输出，然后填充输出梯度来计算输入梯度和过滤器梯度。

1. 批量进行一维卷积：前向传递

当输入具有代表批次大小的第二个维度时，实现前向传递的唯一区别是必须分别填充和计算每个观测值的输出（和之前一样），然后针对结果应用 stack，以获得一批输出。举例来说，conv_1d 函数将变成下面这样。

```
def conv_1d_batch(inp: ndarray,
                  param: ndarray) -> ndarray:
```

```
    outs = [conv_1d(obs, param) for obs in inp]
    return np.stack(outs)
```

2. 批量进行一维卷积：后向传递

后向传递过程与前向传递过程类似：现在只需要使用 for 循环来计算输入梯度，为每个观测值计算该结果，然后执行 stack 操作：

```
# input_grad是包含前述for循环的函数
# 它接受一个一维输入，一个一维过滤器和一个一维output_gradient，并计算input_grad
grads = [_input_grad(inp[i], param, out_grad[i])[1] for i in range(batch_size)]
np.stack(grads)
```

当处理一批观测值时，过滤器的梯度有点不同。这是因为过滤器会与输入中的每个观测值进行卷积，从而连接到输出中的每个观测值。因此，要计算参数梯度，必须循环遍历所有的观测值，并在这个过程中适当增加参数的梯度值。不过，这只需要在代码中添加一个外部 for 循环，计算前面看到的参数梯度：

```
# param：示例中形状为(1,3)的ndarray
# param_grad：和param形状相同的ndarray
# inp：示例中形状为(1,5)的ndarray
# input_pad：形状为(1,7)的ndarray
# output_grad：示例中形状为(1,5)的ndarray
for i in range(inp.shape[0]): # inp.shape[0] = 2
    for o in range(inp.shape[1]): # inp.shape[0] = 5
        for p in range(param.shape[0]): # param.shape[0] = 3
            param_grad[p] += input_pad[i][o+p] * output_grad[i][o]
```

在原始的一维卷积之上添加新维度确实很简单。同样，将其从一维输入扩展到二维输入也很简单。

5.3.4　二维卷积

二维卷积是对一维卷积的直接扩展。从根本上说，在二维场景的每个维度中，输入通过过滤器连接到输出的方式与一维场景相同。因此，前向传递和后向传递的总体步骤均保持不变。

(1) 前向传递

- 适当填充输入。
- 使用所填充的输入和参数来计算输出。

(2) 在后向传递中计算输入梯度

- 适当填充输出梯度。
- 使用所填充的输出梯度以及输入和参数，来计算输入梯度和参数梯度。

(3) 在后向传递中计算参数梯度

- 适当填充输入。
- 遍历所填充的输入的元素，并在这个过程中适当增加参数梯度。

1. 二维卷积：对前向传递进行编码

为了让这一点具体化，可以回顾一维卷积中的相关操作。在一维卷积中，给定前向传递中的输入和参数，用于计算输出的代码如下所示：

```
# input_pad：根据param的大小进行适当填充的输入版本

out = np.zeros_like(inp)

for o in range(out.shape[0]):
    for p in range(param_len):
        out[o] += param[p] * input_pad[o+p]
```

对于二维卷积，只需将其修改为：

```
# input_pad：根据param的大小进行适当填充的输入版本

out = np.zeros_like(inp)

for o_w in range(img_size): # 遍历像高
    for o_h in range(img_size): # 遍历像宽
        for p_w in range(param_size): # 遍历参数宽度
            for p_h in range(param_size): # 遍历参数高度
                out[o_w][o_h] += param[p_w][p_h] * input_pad[o_w+p_w][o_h+p_h]
```

可以看到，后者只是简单地将单个 for 循环变成了两个 for 循环。

当拥有一批图像时，对于二维的扩展也与一维场景类似：如前所述，只是在此处所示的循环外部添加了一层 for 循环。

2. 二维卷积：对后向传递进行编码

可以肯定的是，与前向传递中的一样，可以对后向传递使用与一维场景相同的索引。回想一下，在一维场景中，代码为：

```
input_grad = np.zeros_like(inp)

for o in range(inp.shape[0]):
    for p in range(param_len):
        input_grad[o] += output_pad[o+param_len-p-1] * param[p]
```

在二维场景中，代码也很简单：

```
# output_pad：根据param的大小进行适当填充的输出版本
input_grad = np.zeros_like(inp)
```

```
for i_w in range(img_width):
    for i_h in range(img_height):
        for p_w in range(param_size):
            for p_h in range(param_size):
                input_grad[i_w][i_h] +=
                  output_pad[i_w+param_size-p_w-1][i_h+param_size-p_h-1] \
                  * param[p_w][p_h]
```

注意，输出上的索引与一维场景类似，只是应用到了二维场景中。这是一维场景中的代码：

```
output_pad[i+param_size-p-1] * param[p]
```

这是二维场景中的代码：

```
output_pad[i_w+param_size-p_w-1][i_h+param_size-p_h-1] * param[p_w][p_h]
```

一维场景中的其他情况同样适用于以下两种场景。

(1) 对于一批输入图像，只需对每个观测值执行前面的运算，然后针对结果执行 `stack` 操作。
(2) 对于参数梯度，必须遍历批次中的所有图像，并将每幅图像中的组件添加到参数梯度中的适当位置[8]。

```
input_pad：根据param的大小进行适当填充的输入版本

param_grad = np.zeros_like(param)

for i in range(batch_size): # equal to inp.shape[0]
    for o_w in range(img_size):
        for o_h in range(img_size):
            for p_w in range(param_size):
                for p_h in range(param_size):
                    param_grad[p_w][p_h] += input_pad[i][o_w+p_w][o_h+p_h] \
                    * output_grad[i][o_w][o_h]
```

至此，我们几乎已经为完整的多通道卷积运算编写了代码。当前，代码在二维输入上卷积过滤器并生成二维输出。当然，正如前面所描述的，每个卷积层不仅有沿这两个维度排列的神经元，而且具有一定数量的“通道”，其数量与该层创建的特征图的数量相同。下面便来解决这个最后的挑战。

5.3.5　最后一个元素：通道

在输入和输出都有多个通道的情况下，如何修改前面所写的内容呢？答案很简单，就像之前添加批次时一样，可以在代码中添加两个外部 `for` 循环，一个循环用于输入通道，另一个循环用于输出通道。通过遍历输入通道和输出通道的所有组合，可以根据需要使每个输

注 8：关于这些内容的全部实现，可以从图灵社区下载本章相关资源：ituring.cn/book/2759。——编者注

出特征图成为所有输入特征图的组合。

为此，必须始终将图像表示为三维 ndarray，而不是一直使用的二维数组。下面用一个通道表示黑白图像，并用三个通道表示彩色图像（一个表示图像中每个位置的红色值，一个表示蓝色值，一个表示绿色值）。然后，无论通道数量如何，运算都会按照前面描述的那样进行，从图像中创建许多特征图，每个特征图都是由图像中所有通道（如果处理神经网络中更远的层，则来自上一层中的通道）产生的卷积的组合。

1. 前向传递

鉴于上述情况，给定输入和参数的四维 ndarray，用于计算卷积层输出的完整代码如下所示：

```
def _compute_output_obs(obs: ndarray,
                        param: ndarray) -> ndarray:
    '''
    obs: [channels, img_width, img_height]
    param: [in_channels, out_channels, param_width, param_height]
    '''
    assert_dim(obs, 3)
    assert_dim(param, 4)

    param_size = param.shape[2]
    param_mid = param_size // 2
    obs_pad = _pad_2d_channel(obs, param_mid)

    in_channels = fil.shape[0]
    out_channels = fil.shape[1]
    img_size = obs.shape[1]

    out = np.zeros((out_channels,) + obs.shape[1:])
    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for o_w in range(img_size):
                for o_h in range(img_size):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            out[c_out][o_w][o_h] += \
                              param[c_in][c_out][p_w][p_h]
                              * obs_pad[c_in][o_w+p_w][o_h+p_h]
    return out

def _output(inp: ndarray,
            param: ndarray) -> ndarray:
    '''
    obs: [batch_size, channels, img_width, img_height]
    param: [in_channels, out_channels, param_width, param_height]
    '''
    outs = [_compute_output_obs(obs, param) for obs in inp]

    return np.stack(outs)
```

注意，_pad_2d_channel 是沿通道维度填充输入的函数。

同样，执行计算的实际代码与前面所示的二维场景（没有通道）中的代码相似，但因为在过滤器数组中多了两个维度和 c_out × c_in 个元素，所以现在有了 fil[c_out][c_in][p_w][p_h]，而不是 fil[p_w][p_h]。

2. 后向传递

后向传递与二维场景中的后向传递相似，并且遵循相同的原则。

- 对于输入梯度，分别计算每个观测值的梯度（为此填充输出梯度），然后针对梯度执行 stack 操作。
- 将填充的输出梯度用于参数梯度，但是也会循环遍历观测值，并取每个观测值的适当值来更新参数梯度。

这是计算输出梯度的代码：

```
def _compute_grads_obs(input_obs: ndarray,
                       output_grad_obs: ndarray,
                       param: ndarray) -> ndarray:
    '''
    input_obs: [in_channels, img_width, img_height]
    output_grad_obs: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''
    input_grad = np.zeros_like(input_obs)
    param_size = param.shape[2]
    param_mid = param_size // 2
    img_size = input_obs.shape[1]
    in_channels = input_obs.shape[0]
    out_channels = param.shape[1]
    output_obs_pad = _pad_2d_channel(output_grad_obs, param_mid)

    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for i_w in range(input_obs.shape[1]):
                for i_h in range(input_obs.shape[2]):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            input_grad[c_in][i_w][i_h] += \
                            output_obs_pad[c_out][i_w+param_size-p_w-1]
                            [i_h+param_size-p_h-1] \
                            * param[c_in][c_out][p_w][p_h]
    return input_grad

def _input_grad(inp: ndarray,
                output_grad: ndarray,
                param: ndarray) -> ndarray:

    grads = [_compute_grads_obs(inp[i], output_grad[i], param) for i in
    range(output_grad.shape[0])]

    return np.stack(grads)
```

这是计算参数梯度的代码：

```
def _param_grad(inp: ndarray,
                output_grad: ndarray,
                param: ndarray) -> ndarray:
    '''
    inp: [in_channels, img_width, img_height]
    output_grad_obs: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''
    param_grad = np.zeros_like(param)
    param_size = param.shape[2]
    param_mid = param_size // 2
    img_size = inp.shape[2]
    in_channels = inp.shape[1]
    out_channels = output_grad.shape[1]

    inp_pad = _pad_conv_input(inp, param_mid)
    img_shape = output_grad.shape[2:]

    for i in range(inp.shape[0]):
        for c_in in range(in_channels):
            for c_out in range(out_channels):
                for o_w in range(img_shape[0]):
                    for o_h in range(img_shape[1]):
                        for p_w in range(param_size):
                            for p_h in range(param_size):
                                param_grad[c_in][c_out][p_w][p_h] += \
                                inp_pad[i][c_in][o_w+p_w][o_h+p_h] \
                                * output_grad[i][c_out][o_w][o_h]
    return param_grad
```

_output 函数、_input_grad 函数和 _param_grad 函数正是创建二维卷积运算所需要的，它们最终将构成 CNN 中使用的 Conv2D 层的核心！在将此 Operation 类用于 CNN 之前，还有一些细节问题需要解决。

5.4 使用多通道卷积运算训练CNN

我们还需要实现以下 3 项操作，才能拥有一个有效的 CNN 模型。

1. 必须实现本章前面讨论的 Flatten 运算，这对确保模型能够进行预测很关键。
2. 必须将此 Operation 类以及 Conv2D 运算合并到 Conv2D 层中。
3. 要使这些运算可用，还必须编写更快版本的 Conv2D 运算。这里对此进行概述，附录中的“矩阵链式法则”将详细给出相关信息。

5.4.1 Flatten运算

要完成卷积层，还需要另一个 Operation 类，即 Flatten 运算。卷积运算的输出是每个观测值的三维 ndarray，这 3 个维度分别为 channels、img_height、img_width。但是，除非

将这些数据传递到另一个卷积层，否则首先需要**将每个观测值转换为一个向量**。幸运的是，对于图像中相应位置是否存在特定视觉特征，所有相关的神经元都进行了编码，因此可以简单地将此三维 ndarray“扁平化”（flatten）为一维向量并向前传递，这样做不会有任何问题。这里展示的 Flatten 运算可以实现这一点，这说明在卷积层中（与其他任何层一样），ndarray 的第一个维度始终是批次大小：

```
class Flatten(Operation):
    def __init__(self):
        super().__init__()

    def _output(self) -> ndarray:
        return self.input.reshape(self.input.shape[0], -1)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad.reshape(self.input.shape)
```

那是需要的最后一个 Operation 类，接下来将所有 Operation 类包装在一个 Layer 类中。

5.4.2 完整的Conv2D层

完整的卷积层如下所示：

```
class Conv2D(Layer):

    def __init__(self,
                 out_channels: int,
                 param_size: int,
                 activation: Operation = Sigmoid(),
                 flatten: bool = False) -> None:
        super().__init__()
        self.out_channels = out_channels
        self.param_size = param_size
        self.activation = activation
        self.flatten = flatten

def _setup_layer(self, input_: ndarray) -> ndarray:

    self.params = []
    conv_param = np.random.randn(self.out_channels,
                                 input_.shape[1], # input channels
                                 self.param_size,
                                 self.param_size)
    self.params.append(conv_param)

    self.operations = []
    self.operations.append(Conv2D(conv_param))
    self.operations.append(self.activation)

    if self.flatten:
        self.operations.append(Flatten())

    return None
```

为了进行预测，根据该层输出的预期传递方向，即另一个卷积层（向前传递）或者一个全连接层，可以选择是否在末尾添加 `Flatten` 运算。

关于速度的说明以及替代实现

所有熟悉计算复杂性的人都知道，这部分代码编写起来特别费时间：为了计算参数梯度，需要编写 7 个嵌套 `for` 循环！虽然从零开始编写卷积运算的目的是巩固对 CNN 工作方式的理解，但是可以用完全不同的方式来编写，无须像本章所述的那样分解这个过程，可以将它分解为下面 3 个步骤。

1. 对于输入，从测试集中提取大小为 `filter_height × filter_width` 的 `image_height × image_width × num_channels` 个块。
2. 对于每个块，使用将输入通道连接到输出通道的合适过滤器对块进行点积运算。
3. 堆叠并重塑所有这些点积的结果，从而形成输出。

还有一种更便捷的方法，那就是使用批处理矩阵乘法来表达上述几乎所有的运算，它可以通过 NumPy 库中的 `np.matmul` 函数来实现。附录对如何执行此运算进行了详细的说明，本书的随书文件也对此进行了实现[9]，这些内容说明了可以编写相对较小的 CNN，从而在合理的时间内完成训练。实际上，可以运行实验来查看 CNN 的运行情况！

5.4.3　实验

即使使用通过重塑和 `np.matmul` 函数定义的卷积运算，仅用一个卷积层的话，也大约需要 10 分钟才能完成一轮训练。因此，这里仅限于演示只有一个卷积层的模型，该模型具有 32 个通道（其他数字亦可）：

```
model = NeuralNetwork(
    layers=[Conv2D(out_channels=32,
                   param_size=5,
                   dropout=0.8,
                   weight_init="glorot",
                   flatten=True,
                   activation=Tanh()),
            Dense(neurons=10,
                  activation=Linear())],
            loss = SoftmaxCrossEntropy(),
seed=20190402)
```

注意，该模型在第一层中只有 800 个参数（$32 \times 5 \times 5 = 800$），但是这些参数用于创建 25 088 个神经元（$32 \times 28 \times 28 = 25\ 088$），也可以叫作“学习特征”。相比之下，一个具有 32 个隐藏层的全连接层将有 25 088 个参数（$784 \times 32 = 25\ 088$），同时只有 32 个神经元。

我们通过几百个具有不同学习率的批次来训练该模型，并观察由此产生的验证损失。在这

注 9：也可以从图灵社区下载：ituring.cn/book/2759。——编者注

个过程中，一些简单的试错结果表明，由于第一层是卷积层而不是全连接层，因此 0.01 的学习率要优于 0.1 的学习率。使用优化器 SGDMomentum(lr = 0.01, momentum=0.9)，将这个神经网络训练一轮，可以得到以下结果：

```
Validation accuracy after 100 batches is 79.65%
Validation accuracy after 200 batches is 86.25%
Validation accuracy after 300 batches is 85.47%
Validation accuracy after 400 batches is 87.27%
Validation accuracy after 500 batches is 88.93%
Validation accuracy after 600 batches is 88.25%
Validation accuracy after 700 batches is 89.91%
Validation accuracy after 800 batches is 89.59%
Validation accuracy after 900 batches is 89.96%
Validation loss after 1 epochs is 3.453

Model validation accuracy after 1 epoch is 90.50%
```

这表明确实可以从零开始训练 CNN，而且只需经过一次训练，就可以在 MNIST 上达到 90% 以上的准确度！

5.5 小结

本章主要介绍卷积神经网络（CNN），首先概述基本概念及 CNN 与全连接神经网络的异同，然后深入介绍它的工作方式，并使用 Python 从零开始实现核心的多通道卷积运算。

总体上看，在创建神经元数量方面，卷积层比目前所看到的全连接层大约多出一个数量级，但由于每个神经元都位于全连接层中，因此它们只是上一层的部分特征，而不是所有特征的组合。这些神经元实际上被分为“特征图”，每个特征图都代表在图像中的给定位置上是否存在特定的视觉特征，相当于深度卷积神经网络中的视觉特征组合，这些特征图统称为卷积 Layer 类的“通道”。

尽管与 Dense 层所涉及的 Operation 类存在差异，但是卷积运算与前面的其他 ParamOperation 都适用于同一模板，具有下面两个特征。

- 它具有 _output 方法，根据给定的输入和参数来计算输出。
- 它具有 _input_grad 方法和 _param_grad 方法，给定与 Operation 类的 output 具有相同形状的 output_grad，可以分别计算与输入和参数具有相同形状的梯度。

不同之处在于，_input、output 和 param 现在是四维 ndarray，而它们在全连接层的情况下是二维 ndarray。

这些内容为 CNN 奠定了坚实的基础，包括将来的学习以及相关应用程序的使用。接下来将介绍另一种常见的高级神经网络架构，即 RNN（循环神经网络），这种神经网络不再简单地处理房屋示例和图像示例中的非顺序批处理数据，而是处理按顺序出现的数据。加油！

第 6 章 RNN

本章将介绍 RNN，即**循环神经网络**（recurrent neural network），它是一种用于处理数据序列的神经网络架构。目前所介绍的神经网络将其接收到的每一批数据都视为一组独立的观测值，也就是说，无论是第 1 ～ 4 章中的全连接神经网络，还是第 5 章中的 CNN，其中都不存在一些 MNIST 数字在其他数字之前或之后到达的概念。然而，不管是可能在工业或金融环境中处理的时间序列数据，还是按字符、单词、词性等顺序排列的语言数据，许多数据在本质上是有序的。RNN 旨在学习如何接收这些数据**序列**，并返回正确的预测作为输出，比如第二天金融资产的价格或者句子中将出现的下一个单词。

与全连接神经网络相比，处理有序数据需要完成 3 处修改。第一，在要输入到神经网络的 `ndarray` 中“增加一个新维度”。以前，提供给神经网络的数据本质上是二维的，即每个 `ndarray` 用一维表示观测值，用另一维表示特征数[1]。另一种思考方式是，**每个观测值都是一维向量**。在使用 RNN 时，每个输入仍将具有一个维度来表示观测值，但每个观测值都将被表示为一个二维 `ndarray`：一个维度代表数据序列的长度，另一个维度表示每个序列元素中存在的特征数。因此，RNN 的整体输入将是一批序列，即包含形状为 `[batch_size, sequence_length, num_features]` 的三维 `ndarray`。

第二，要处理这种新的三维输入，必须使用一种新型的神经网络架构，这将是本章的重点。但是，本章将最先讨论第三个变化：必须使用具有不同抽象的框架来处理这种新的数据形式。这是因为在全连接神经网络和 CNN 中，即使每个“运算”实际上代表了许多单独的加法和乘法（例如矩阵乘法或卷积），也可以被描述为一个“微型工厂”，在前向传递

注 1：我们碰巧发现沿行排列观测值并且沿列排列特征很方便，但不一定非要这样排列数据。然而，数据确实必须是二维的。

和后向传递中，它们都需要一个 `ndarray` 作为输入，并生成一个 `ndarray` 作为输出，还有可能使用代表运算参数的另一个 `ndarray` 作为这些计算的一部分。事实证明，RNN 并不能以这种方式实现。在深入了解原因之前，先来思考一下：神经网络架构的哪些特征会导致目前所构建的框架出现问题？虽然这个问题的答案很有启发性，但完整的解决方案涉及实现细节，这超出了本书的讨论范围[2]。要剖析这一点，需要揭示目前所用框架的一个关键限制。

6.1　关键限制：处理分支

事实证明，目前的框架无法使用如图 6-1 所示的计算图来训练模型。

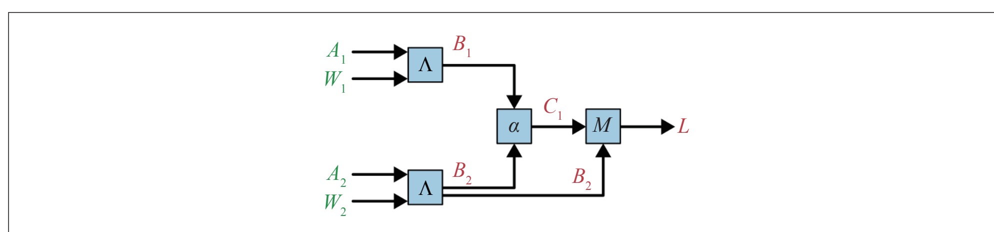


图 6-1：导致运算框架失效的计算图：相同的量在前向传递中多次重复，这意味着不能像以前一样简单地在后向传递中按顺序向后发送梯度

这是怎么回事？将前向传递转换为代码似乎还好，如以下代码所示。注意，这里的 `Add` 运算和 `Multiply` 运算仅供说明之用。

```
a1 = torch.randn(3,3)
w1 = torch.randn(3,3)

a2 = torch.randn(3,3)
w2 = torch.randn(3,3)

w3 = torch.randn(3,3)

# 运算
wm1 = WeightMultiply(w1)
wm2 = WeightMultiply(w2)
add2 = Add(2, 1)
mult3 = Multiply(2, 1)

b1 = wm1.forward(a1)
b2 = wm2.forward(a2)
c1 = add2.forward((b1, b2))
L = mult3.forward((c1, b2))
```

注 2：本书不会涉及相关内容，但不排除在后续版本中有所更新。

当开始后向传递时，麻烦就来了。假设要使用常规链式法则逻辑来计算 L 相对于 W_1 的导数。以前，只需以相反的顺序在每个运算上调用 backward 方法即可。在这里，由于在前向传递过程中重用了 B_2，因此该方法无效。如果对 mult3 调用 backward 方法，则每个输入（C_1 和 B_2）都将具有梯度。但是，如果随后对 add2 调用 backward 方法，那么将无法仅输入 C_1 的梯度：还必须以某种方式输入 B_2 的梯度，因为这也会影响损失 L。因此，要正确对该计算图执行后向传递，不能完全只按照相反的顺序执行运算。必须手动编写如下内容：

```
c1_grad, b2_grad_1 = mult3.backward(L_grad)

b1_grad, b2_grad_2 = add2.backward(c1_grad)

# 将这些梯度组合起来，表示B2在前向传递中使用了两次
b2_grad = b2_grad_1 + b2_grad_2

a2_grad = wm2.backward(b2_grad)

a1_grad = wm1.backward(b1_grad)
```

在这一点上，最好完全跳过使用 Operation 类。可以像第 2 章中那样，简单地保存在前向传递中计算出的所有量，并在后向传递中重用它们！另外，通过手动定义要在神经网络的前向传递和后向传递中完成的各个计算，可以编写任意复杂的神经网络，这和在第 2 章中实现双层神经网络的后向传递时所涉及的 17 个独立运算是一样的，6.5 节会执行类似的运算。我们试图通过 Operation 类来构建一个灵活的框架，该框架能够用高级术语描述神经网络，并让所有的低级计算“只负责工作”。虽然这个框架说明了许多关于神经网络的关键概念，但是它也有局限性。

这个问题有一个优雅的解决方案：**自动微分**（automatic differentiation），这是一种完全不同的神经网络实现方法[3]。考虑到构建一个功能齐全的自动微分框架本身就需要花费几章的篇幅，这里只涉及与其工作机制相关的概念，并不会介绍过多内容。此外，第 7 章在介绍 PyTorch 时会提到如何使用高性能自动微分框架。尽管如此，自动微分仍然是一个重要的概念，在深入学习 RNN 之前，可以从基本原理上进行理解。这里将为它设计一个基本框架，并展示它如何解决前面示例中描述的在前向传递期间重用对象的问题。

6.2 自动微分

正如前面所介绍的，必须针对某些神经网络架构训练模型，而目前使用的 Operation 框架在计算输出相对于输入的梯度时并不乐观。自动微分可以通过完全不同的路径计算这些梯

注 3：这个问题还有一个解决方案，这是 Daniel Sabinasz 在他的博客“deep ideas”上分享的：他将运算表示为一个示意图，然后使用广度优先搜索，在后向传递中按照正确的顺序计算梯度，最终构建一个模仿 TensorFlow 的框架。他发表的关于如何做到这一点的博客文章结构明晰，表述清楚。

度：不是将 Operation 类作为构成神经网络的基本单位，而是定义一个类，该类包装了数据本身，并可以让数据跟踪对其执行的运算，这样在涉及不同运算时，数据就可以不断累积梯度。为了更好地理解这种“梯度累积”的工作原理，接下来开始对其进行编码[4]。

编码梯度累积

为了自动跟踪梯度，必须重写对数据执行基本运算的 Python 方法。在 Python 中，使用 + 或 - 等运算符实际上会调用 _add_ 或 _sub_ 之类的基础隐藏方法。例如，这是 + 运算符的工作方式：

```
a = array([3,3])
print("Addition using '__add__':", a.__add__(4))
print("Addition using '+':", a + 4)

Addition using '__add__': [7 7]
Addition using '+': [7 7]
```

可以利用这个优势编写一个类，该类包装典型的 Python“数字”（float 或 int），并重写 add 方法和 mul 方法：

```
Numberable = Union[float, int]

def ensure_number(num: Numberable) -> NumberWithGrad:
    if isinstance(num, NumberWithGrad):
        return num
    else:
        return NumberWithGrad(num)

class NumberWithGrad(object):

    def __init__(self,
                 num: Numberable,
                 depends_on: List[Numberable] = None,
                 creation_op: str = ''):
        self.num = num
        self.grad = None
        self.depends_on = depends_on or []
        self.creation_op = creation_op

    def __add__(self,
                other: Numberable) -> NumberWithGrad:
        return NumberWithGrad(self.num + ensure_number(other).num,
                              depends_on = [self, ensure_number(other)],
                              creation_op = 'add')

    def __mul__(self,
                other: Numberable = None) -> NumberWithGrad:
```

注 4：要深入理解如何实现自动微分，请参阅 Andrew Trask 所著的《深度学习图解》。

```
        return NumberWithGrad(self.num * ensure_number(other).num,
                              depends_on = [self, ensure_number(other)],
                              creation_op = 'mul')

    def backward(self, backward_grad: Numberable = None) -> None:
        if backward_grad is None: # first time calling backward
            self.grad = 1
        else:
            # 这些行允许累积梯度。
            # 如果梯度尚不存在，就将其设置为backward_grad
            if self.grad is None:
                self.grad = backward_grad
            # 否则，只需向现有梯度添加backward_grad
            else:
                self.grad += backward_grad

        if self.creation_op == "add":
            # 由于增加这两个元素中的任意一个都会使输出增加相同的量，
            # 因此只需向后发送self.grad
            self.depends_on[0].backward(self.grad)
            self.depends_on[1].backward(self.grad)

        if self.creation_op == "mul":

            # 计算关于第1个元素的导数
            new = self.depends_on[1] * self.grad
            # 向后发送关于该元素的导数
            self.depends_on[0].backward(new.num)

            # 计算关于第2个元素的导数
            new = self.depends_on[0] * self.grad
            # 向后发送关于该元素的导数
            self.depends_on[1].backward(new.num)
```

这里执行了很多运算。让我们仔细研究 `NumberWithGrad` 类的工作方式。回想一下，这样一个类的目标是能够编写简单的运算并自动计算梯度。假设像下面这样编写代码：

```
a = NumberWithGrad(3)

b = a * 4
c = b + 5
```

此时，将 `a` 增加 ϵ 后，`c` 的值会增加多少？很明显，`c` 会增加 $4 \times \epsilon$。实际上，使用前面的类，如果首先像这样编写：

```
c.backward()
```

然后，无须编写 `for` 循环遍历 `Operation` 类或其他运算，便可以得到下面的结果：

```
print(a.grad)
```

```
4
```

这是什么原因呢？上述类的基本思想是，每当对 NumberWithGrad 类执行 + 运算或 * 运算时，都会创建一个新的 NumberWithGrad 类，并将第一个 NumberWithGrad 类作为依赖项。然后，就像前面调用 c 一样，在针对 NumberWithGrad 类调用 backward 方法时，将自动计算用于创建 c 的所有 NumberWithGrad 类的所有梯度。因此，这个过程不仅计算了 a 的梯度，也计算了 b 的梯度：

```
print(b.grad)
```

```
1
```

然而，这个框架的真正优势在于，**它允许 NumberWithGrad 类累积梯度**，从而使这些梯度在一系列计算中多次重用，而最终仍然可以得到正确的梯度。这里将用之前介绍过的一系列相同的运算，通过在一系列计算中多次使用 NumberWithGrad 类来说明，同时详细解释它的工作方式。

1. 自动微分详解

下面是一系列运算，其中 a 被多次重用：

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = c * (a + 2)
```

如果执行这些运算，那么可以算出 d = 75，但是真正的问题是：当 a 的值增加后，d 的值会增加多少？可以用数学方法得出这个问题的答案：

$$d = (4a+3) \times (a+2) = 4a^2 + 11a + 6$$

使用微积分中的**幂律**（power rule）：

$$\frac{\partial d}{\partial a} = 8a + 11$$

因此，对于 $a = 3$ ，该导数的值应为 $8 \times 3 + 11 = 35$ 。用数字进行确认：

```
def forward(num: int):
    b = num * 4
    c = b + 3
    return c * (num + 2)

print(round(forward(3.01) - forward(2.99)) / 0.02), 3)
```

```
35.0
```

可以看到，在使用自动微分框架计算梯度时，能得到相同的结果。

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e.backward()

print(a.grad)

35
```

2. 原理解释

如前所述，自动微分的目标是让**数据对象**（数字、ndarray、Tensor 类等）成为分析的基本单位，而不是用以前的 Operation 类。

所有自动微分技术都具有以下共同点。

- 每种技术都包含一个类，该类包装正在计算的实际数据，这里使用 NumberWithGrad 类来包装 float 和 int。在 PyTorch 中，类似的类叫作 Tensor。
- 重新定义了加法、乘法和矩阵乘法之类的常用运算，这样它们始终返回这个类的成员。在上述情况下，要确保对两个 NumberWithGrad 类，或者一个 NumberWithGrad 类和一个 float（或 int）执行加法。
- 给定在前向传递中会发生的情况，NumberWithGrad 类必须包含有关如何计算梯度的信息。以前是通过在类中包含 creation_op 参数来完成此操作，该参数仅记录 NumberWithGrad 类的创建方式。
- 在后向传递中，使用基础数据类型（而不是包装器）将梯度向后传递。这意味着梯度的类型是 float 和 int，而不是 NumberWithGrad 类。
- 如本节开头所述，自动微分使我们能够重用在前向传递过程中计算出的量。前面的示例使用了两次 a 而没有出现问题，关键是以下几行代码：

  ```
  if self.grad is None:
      self.grad = backward_grad
  else:
      self.grad += backward_grad
  ```

  这些代码行表示，当接收到新的梯度 backward_grad 时（一个 NumberWithGrad 类），应该将 NumberWithGrad 类的梯度初始化为这个值，或者简单地将该数值添加到 NumberWithGrad 类的现有梯度中。当相关对象在模型中被重用时，这些代码可以让 NumberWithGrad 类累积梯度。

以上就是自动微分的全部内容。因为它需要在前向传递过程中重用某些量才能进行预测，所以接下来看一下引发这种话题的模型结构。

6.3 RNN的动机

正如本章开头所讨论的那样，RNN 旨在处理序列中出现的数据：现在，每个观测值不再是包含 n 个特征的向量，而是一个 n 个特征乘以 t 个时间步长的二维数组，如图 6-2 所示。

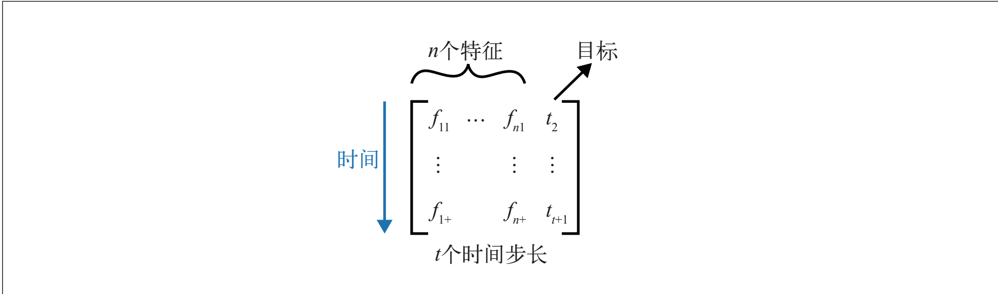


图 6-2：序列数据：在 t 个时间步长中，每一个时间步长都有 n 个特征

接下来的几节将说明 RNN 如何容纳这种形式的数据，但首先来看一下使用 RNN 的动机。仅使用常规前馈神经网络处理此类数据的局限性是什么？尝试将每个时间步长都表示为一组独立的特征。例如，一个观测值可能具有来自时间 $t=1$ 的特征，而目标值则来自时间 $t=2$，下一个观测值可能具有来自时间 $t=2$ 的特征，而目标值则来自时间 $t=3$，以此类推。如果想使用来自多个时间步长的数据进行预测，则可以使用 $t=1$ 和 $t=2$ 的特征来预测 $t=3$ 的目标，使用 $t=2$ 和 $t=3$ 的特征来预测 $t=4$ 的目标，以此类推。

然而，将每个时间步长视为独立特征忽略了数据是按顺序排列的这一事实。在理想情况下，如何利用数据的顺序性做出更好的预测呢？解决方案可以参考下面几个步骤。

1. 使用时间步长 $t=1$ 中的特征为 $t=1$ 处的相应目标进行预测。
2. 使用时间步长 $t=2$ 中的特征和 $t=1$ 中的信息，包括 $t=1$ 处的目标值，来进行 $t=2$ 处的预测。
3. 使用时间步长 $t=3$ 中的特征以及 $t=1$ 和 $t=2$ 中的累积信息来进行 $t=3$ 处的预测。
4. 以此类推，在每个步骤中，使用所有先前时间步长中的信息进行预测。

为了做到这一点，还需要通过神经网络一次传递一个序列元素，首先传递第一个时间步长的数据，然后传递下一个时间步长的数据，以此类推。此外，当新的序列元素通过时，神经网络可以“累积”关于它以前所看到的信息。本章随后将详细讨论 RNN 如何实现这一点，那时可以看到，虽然 RNN 有多个变体，但它们在顺序处理数据的方式上都有一个共同的基础结构。本章将花大部分篇幅讨论这个结构，并在最后讨论各个变体的不同之处。

6.4 RNN简介

现在从总体上回顾如何通过前馈神经网络传递数据，进而开始对 RNN 的讨论。在这种类型的神经网络中，数据通过一系列层进行传递。对于单个观测值，每一层的输出都是神经网络对这一层中该观测值的**表征**（representation）。在第一层之后，该表征包含由原始特征所组合的特征。在第二层之后，它包含这些表征的组合，也可以说原始特征的“特征的特征”，以此类推，神经网络中的其他后续的层也是这种情况。然后，在每次前向传递之后，神经网络在其每一层的输出中都包含原始观测值的许多表征，如图 6-3 所示。

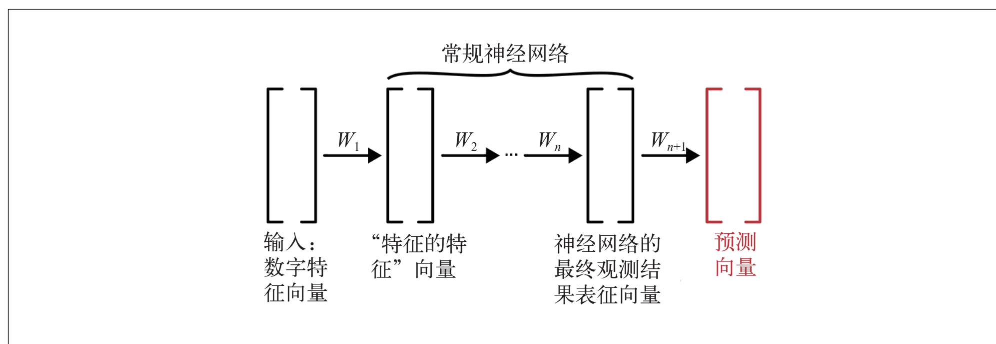


图 6-3：常规神经网络将观测值向前传递并在每一层后将其转换成不同的表征

但是，当下一组观测值通过神经网络时，这些表征将被丢弃。RNN 及其所有变体的关键创新是**将这些表征与下一组观测值一起传递回神经网络**。图 6-4 是这个过程的示意图，文字表述如下。

1. 在第一个时间步长中，传递 $t=1$ 处的观测值（可能还带有随机初始化的表征），然后输出 $t=1$ 的预测值以及每一层的表征。
2. 在第二个时间步长中，传递 $t=2$ 处的观测值以及在第一个时间步长中计算的表征（同样，这只是神经网络各层的输出），并以某种方式将它们进行组合。注意，这个组合步骤将体现各种 RNN 变体的差异。使用这两条信息来输出 $t=2$ 的预测值以及每一层中被更新的表征，它们现在是同时在 $t=1$ 处和 $t=2$ 处所传入的输入的函数。
3. 在第三个时间步长中，传递 $t=3$ 处的观测值和包含来自 $t=1$ 和 $t=2$ 的信息的表征，并使用此信息对 $t=3$ 进行预测，同时更新每层中的附加表征，这些表征现在包含了第一个时间步长到第三个时间步长中的信息。

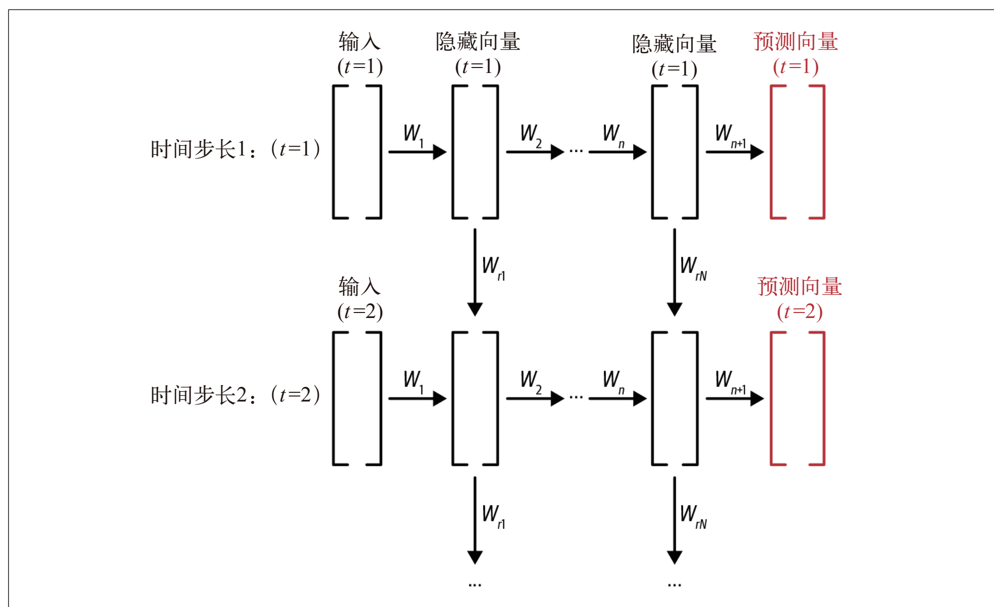


图 6-4：RNN 将每一层的表征传递到下一个时间步长

可以看到，每层都有一个“持久化”的表征，随着时间的推移，它会随着新观测值的传递而不断更新。事实上，这就是 RNN 不适合前几章编写的 `Operation` 框架的原因：为了通过 RNN 对单个数据序列进行一组预测，代表每一层状态的 `ndarray` 会不断更新并多次重用。由于无法使用第 5 章中的框架，因此必须从基本原则出发，明确处理 RNN 所需构建的类。

6.4.1 RNN的第一个类：`RNNLayer`

根据对 RNN 工作方式的描述，可以知道至少需要一个 `RNNLayer` 类，该类一次将一个数据序列传递给一个序列元素。现在来详细了解此类的工作方式，正如本章提到的那样，RNN 将处理包含二维观测值的数据，涉及 `sequence_length` 和 `num_features`。另外，由于以批量方式向前传递数据在计算上更加高效，因此 `RNNLayer` 类将必须接收一个三维 `ndarray`（`batch_size`、`sequence_length` 和 `num_features`）。但是，6.3 节刚开始就提到过，我们希望一次传给 `RNNLayer` 类一个序列的数据，那么如果输入的 `data` 是 `[batch_size, sequence_length, num_features]`，如何实现这一点？可以参考下面的步骤。

1. 从第二个轴中选择一个以 `data[:, 0, :]` 开始的二维数组。这个 `ndarray` 的形状为 `[batch_size, num_features]`。
2. 为 `RNNLayer` 类初始化一个“隐藏状态”，此状态将随着每个序列元素的传入而不断更新，这里形状为 `[batch_size, hidden_size]`。这个 `ndarray` 表示该层在先前时间步长中已传递的有关数据的“累积信息”。

3. 将这两个 ndarray 向前传递到该层的第一个时间步长。最终设计 RNNLayer 类来输出 ndarray（与输入的维度不同），这和常规的 Dense 层是一样的，因此输出将具有形状 [batch_size, num_outputs]。此外，为每个观测值都更新神经网络的表征：在每个时间步长中，RNNLayer 类还应该输出一个形状为 [batch_size, hidden_size] 的 ndarray。
4. 从 data: data[:, 1, :] 中选择下一个二维数组。
5. 将该数据和在第一个时间步长中输出的 RNN 表征值传递到该层的第二个时间步长，从而获得形状为 [batch_size, num_outputs] 的另一个输出，以及形状为 [batch_size, hidden_size] 的更新表征。
6. 继续执行上述操作，直到 sequence_length 个时间步长都已通过该层。然后将所有结果串联在一起，来获取该层的输出，其形状为 [batch_size, sequence_length, num_outputs]。

上面几个步骤有助于理解 RNNLayer 类的工作方式，而且在编写代码时也可以巩固这种理解。但它暗示需要另一个类来处理接收到的数据并在每个时间步长中更新层的隐藏状态。这时需要使用 RNNNode 类，也就是要介绍的下一个类。

6.4.2　RNN的第二个类：RNNNode

RNNNode 类应该具有包含以下输入和输出的 forward 方法。

1. 两个作为输入的 ndarray。
   - 一个用于输入到神经网络的数据，形状为 [batch_size, num_features]。
   - 一个用于该时间步长的观测值表征，形状为 [batch_size, hidden_size]。
2. 两个作为输出的 ndarray。
   - 一个用于该时间步长的神经网络输出，形状为 [batch_size, num_outputs]。
   - 一个用于该时间步长中观测值的更新表征，形状为 [batch_size, hidden_size]。

接下来展示如何将 RNNNode 类和 RNNLayer 类整合在一起。

6.4.3　整合RNNNode类和RNNLayer类

RNNLayer 类将包装一个 RNNNode 列表，并且至少将包含具有以下输入和输出的 forward 方法。

1. 输入：一批观测值序列，形状为 [batch_size, sequence_length, num_features]。
2. 输出：这些序列的神经网络输出，形状为 [batch_size, sequence_length, num_outputs]。

图 6-5 显示了数据通过 RNN 向前移动的顺序，该 RNN 具有两个 RNNLayer 类，其中各包含 5 个 RNNNode 类。在每个时间步长中，尺寸为 feature_size 的初始输入依次通过每个 RNNLayer 类中的第一个 RNNNode 类向前传递，神经网络最终在该时间步长输出维度为 output_size 的预测值。此外，每个 RNNNode 类都将“隐藏状态”传递到每层中的下一个 RNNNode 类。一旦 5 个时间步长中的数据都通过了所有层，就将获得形状为 [5, output_size] 的最终预测集，其中 output_size 应具有与目标相同的维度。然后将这些预测与目标进行比较，并计算出

损失梯度，从而启动后向传递。图 6-5 对此进行了总结，显示了数据流经 5×2 个 `RNNNode` 类的顺序（从 1 到 10）。

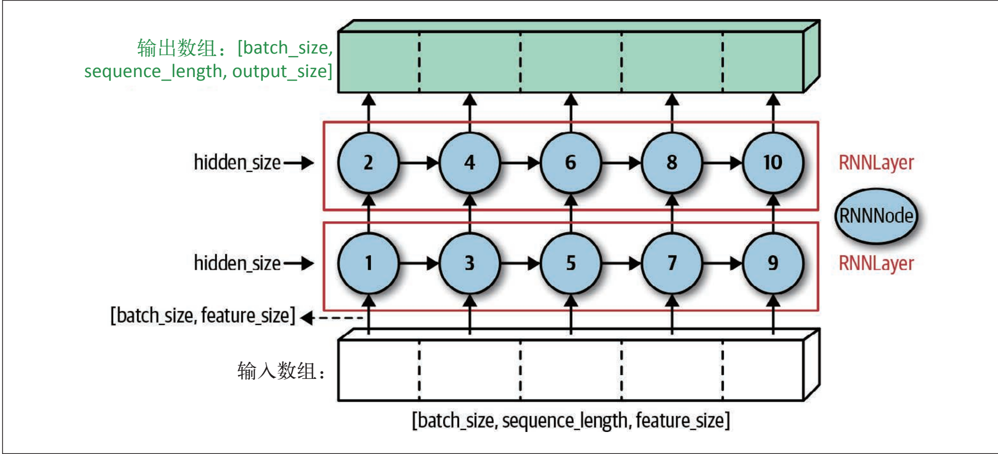


图 6-5：数据流经 RNN 的顺序，该 RNN 有两层，用于处理长度为 5 的序列

此外，数据也可以按图 6-6 所示的顺序流经 RNN。无论顺序如何，都会发生以下 3 种情况。

1. 每层的数据处理都发生在下一层之前的给定时间步长。例如，在图 6-5 中，2 不能在 1 之前发生，4 不能在 3 之前发生。
2. 同样，每一层都必须按顺序处理所有时间步长。例如，在图 6-5 中，4 不能在 2 之前发生，3 不能在 1 之前发生。
3. 最后一层必须为每个观测值都输出维度 `feature_size`。

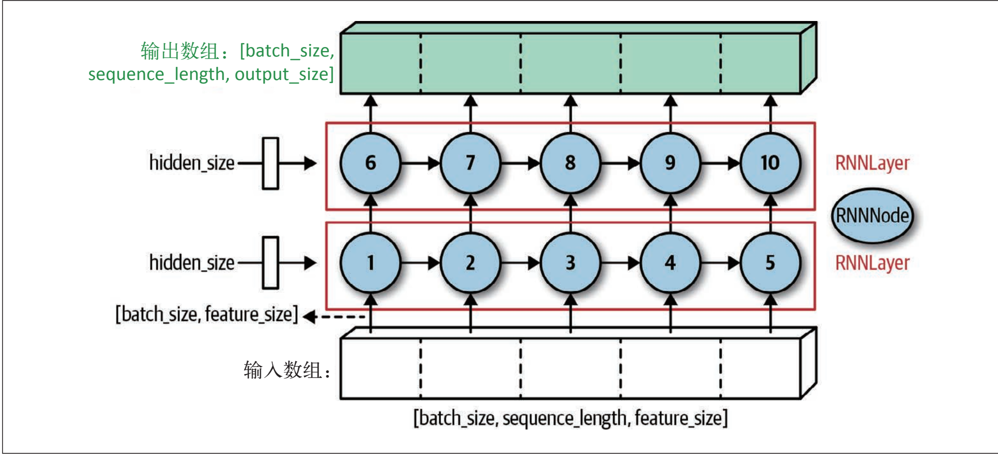


图 6-6：数据按照另一顺序在前向传递过程中流经同一个 RNN

以上就是 RNN 前向传递的工作方式。后向传递又是怎样的呢？

6.4.4　后向传递

通过 RNN 进行的反向传播通常会描述为一种单独的算法，称作“基于时间的反向传播”[5]。尽管这确实描述了反向传播期间发生的情况，但听起来比实际情况要复杂得多。记住关于数据如何通过 RNN 向前流动的解释部分，可以用这种方式描述后向传递的过程：通过将梯度向后传递给神经网络（采用与在前向传递中传递输入相反的顺序），从而将数据向后传递给 RNN。实际上，这和常规前馈神经网络中的操作一样。

在前向传递过程中，查看图 6-5 和 6-6 中的示意图，可以得出以下 5 个结论。

1. 从一批观测值开始，每个观测值的形状为 `[feature_size，sequence_length]`。
2. 这些输入被分解为单独的 `sequence_length` 元素，并一次传递到神经网络中。
3. 每个元素都通过所有层，最终转换成大小为 `output_size` 的输出。
4. 同时，层在下一个时间步长中将隐藏状态向前传递到层的计算中。
5. 对于所有 `sequence_length` 时间步长，这个过程都持续进行，从而得出大小为 `[output_size，sequence_length]` 的总输出。

反向传播的工作方式与此完全相同，但过程相反，具体如下。

1. 从形状为 `[output_size， sequence_length]` 的梯度开始，该梯度表示输出的每个元素（大小同样为 `[output_size，sequence_length]`）最终对为该批观测值所计算的损失的影响程度。
2. 这些梯度被分解为单独的 `sequence_length` 元素，并**以相反的顺序后向传递**通过各层。
3. 单个元素的梯度后向传递通过所有层。
4. 同时，各层**将该时间步长中相对于隐藏状态的损失梯度**向后传递到各层先前的时间步长的计算中。
5. 对于所有 `sequence_length` 时间步长，这个过程都持续进行，直到将梯度向后传递到神经网络中的每一层，从而可以计算相对于每个权重的损失梯度，就像在常规前馈神经网络中所做的那样。

图 6-7 突出展示了后向传递和前向传递之间的这种并行性，以及数据在后向传递中流经 RNN 的方式。当然，可以看到，垂直方向上的箭头与图 6-5 中的相反。

注 5：backpropagation through time，BPTT。——译者注

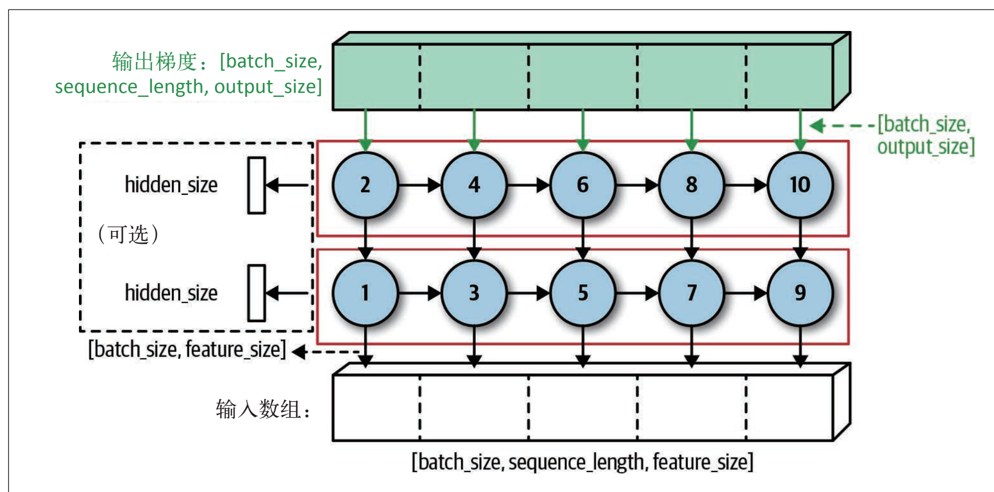


图 6-7：在后向传递中，RNN 以与前向传递数据相反的方向传递数据

这表明，总体而言，RNNLayer 类的前向传递和后向传递与它们在常规神经网络层中的非常相似：它们都接收某种形状的 ndarray 作为输入，输出另一种形状的 ndarray，并且在后向传递中接收到的输出梯度与输出形状相同，同时产生与输入形状相同的输入梯度。但是，与其他层相比，RNNLayer 类中处理权重梯度的方式存在一个关键差异，在开始编写全部代码之前，我们将简要讨论这一点。

RNN 中权重的累积梯度

就像在常规神经网络中一样，RNN 中的每一层也都将具有一组权重。这意味着相同的权重集将在所有 sequence_length 时间步长上影响层的输出。因此，在反向传播期间，相同的权重集将接收 sequence_length 个梯度。以图 6-7 所示的反向传播为例，在标记为“1”的圆圈中，第二层将在最后一个时间步长处接收到一个梯度，而在标记为“3”的圆圈中，该层将接收倒数第二个时间步长的梯度，两者将由相同的权重集驱动。因此，在反向传播过程中，必须在一系列时间步长上**累积权重的梯度**，这意味着无论选择如何存储权重，都必须使用类似以下的方法来更新它们的梯度：

```
weight_grad += grad_from_time_step
```

这与 Dense 层和 Conv2D 层不同，这两种层仅将参数存储在 param_grad 参数中。

前面已经列出了 RNN 的工作方式以及实现它们所要构建的类，接下来具体看看实现细节。

6.5 RNN：代码

下面从实现 RNN 的几种方法开始，这些方法与本书介绍的其他神经网络的方法类似。

1. RNN 仍将通过一系列层向前传递数据，这些层在前向传递过程中向前发送输出，在后向传递过程中向后发送梯度。因此，无论 NeuralNetwork 类最终如何替换，都仍然会有一个包含 RNNLayer 列表的 layers 属性，前向传递将由如下代码实现。

```
def forward(self, x_batch: ndarray) -> ndarray:

    assert_dim(ndarray, 3)

    x_out = x_batch
    for layer in self.layers:

        x_out = layer.forward(x_out)

    return x_out
```

2. RNN 的 Loss 与之前相同：最后的 Layer 类生成一个返回 ndarray 的 output，并与 y_batch 进行比较。然后计算出一个值，并以与 output 相同的形状返回该值相对于 Loss 输入的梯度。为了与形状为 [batch_size， sequence_length， feature_size] 的 ndarray 配合使用，必须修改 softmax 函数，但这个问题可以解决。
3. Trainer 几乎是相同的：循环浏览训练数据，选择多个批次的输入数据和输出数据，并不断将它们输入到模型中并生成损失值，以此判断在每批数据输入完毕后模型是否在学习和更新权重。
4. Optimizer 类也保持不变。正如将看到的，必须在每个时间步长中更新提取 params 参数和 param_grads 参数的方式，但是“更新规则”（在类的 _update_rule 函数中捕获的规则）保持不变。

Layer 类本身就很有趣，接下来详细讨论。

6.5.1 RNNLayer类

前面提到，Layer 类包含一组 Operation 类，这些 Operation 类向前传递数据并向后发送梯度。然而，RNNLayer 类完全不同，它们现在必须保持“隐藏状态”，该状态将随着新数据的输入不断更新，并在每个时间步长中以某种方式与这些数据进行“合并”。至于具体的工作方式，可以参考图 6-5 和图 6-6：每个 RNNLayer 类都应该具有一个 RNNNode 列表作为属性，然后层的 input 中的每个序列元素都应该以一次一个元素的方式通过每个 RNNNode 列表。每个 RNNNode 列表都将接收该序列元素以及该层的“隐藏状态”，在这个时间步长中为该层生成输出，并更新该层的隐藏状态。

为了清楚地展示这个过程，现在开始对其进行编码，其中将依次介绍如何初始化 RNNLayer 类以及如何在前向传递（或后向传递）过程中向前（或向后）发送数据。

1. 初始化

每个 RNNLayer 类都将用以下内容开头。

- 一个 int 类型的 hidden_size
- 一个 int 类型的 output_size
- 一个形状为 [1, hidden_size] 的 ndarray start_H，表示层的隐藏状态

此外，就像在常规神经网络中一样，当初始化层时，设置 self.first = True。在第一次将数据传递到 forward 方法时，将接收到的 ndarray 传递给 _init_params 方法，初始化参数并设置 self.first = False。

完成层的初始化之后，接下来看一下如何发送数据。

2. forward 方法

forward 方法的主要部分将包括接收一个形状为 [batch_size, sequence_length, feature_size] 的 ndarray x_seq_in，并将其按顺序传递给层的 RNNNode 类。在以下代码中，self.nodes 是该层的 RNNNode 类，H_in 则是该层的隐藏状态：

```
sequence_length = x_seq_in.shape[1]

x_seq_out = np.zeros((batch_size, sequence_length, self.output_size))

for t in range(sequence_length):

    x_in = x_seq_in[:, t, :]

    y_out, H_in = self.nodes[t].forward(x_in, H_in, self.params)

    x_seq_out[:, t, :] = y_out
```

关于隐藏状态 H_in，需要注意一点：RNNLayer 类的隐藏状态通常以向量表示，但是每个 RNNNode 类中的运算都要求隐藏状态是一个大小为 [batch_size, hidden_size] 的 ndarray。因此，在开始每次前向传递时，都只是“重复”隐藏状态：

```
batch_size = x_seq_in.shape[0]

H_in = np.copy(self.start_H)

H_in = np.repeat(H_in, batch_size, axis=0)
```

在前向传递之后，取各批次观测值的平均值，来获取该层更新后的隐藏状态：

```
self.start_H = H_in.mean(axis=0, keepdims=True)
```

此外，从代码中可以看到，RNNNode 类必须有一个 forward 方法，该方法接受如下形状的两个数组。

- [batch_size, feature_size]
- [batch_size, hidden_size]

同时，会返回如下形状的两个数组。

- [batch_size, output_size]
- [batch_size, hidden_size]

后文会介绍 RNNNode 类及其变体，不过在开始之前，先来看一下 RNNLayer 类的 backward 方法。

3. backward 方法

由于 forward 方法输出 x_seq_out，因此 backward 方法将接收与 x_seq_out 形状相同的梯度，即 x_seq_out_grad。与 forward 方法的移动方向相反，这里通过 RNNNode 类向后传递这个梯度，最终返回形状为 [batch_size, sequence_length, self.feature_size] 的 x_seq_in_grad 作为整层的梯度：

```
h_in_grad = np.zeros((batch_size, self.hidden_size))

sequence_length = x_seq_out_grad.shape[1]

x_seq_in_grad = np.zeros((batch_size, sequence_length, self.feature_size))

for t in reversed(range(sequence_length)):

    x_out_grad = x_seq_out_grad[:, t, :]

    grad_out, h_in_grad = \
        self.nodes[t].backward(x_out_grad, h_in_grad, self.params)

    x_seq_in_grad[:, t, :] = grad_out
```

由此可见，为了遵循模式，RNNNode 类应该有一个 backward 方法，它与 forward 方法相反，该方法接收如下形状的两个数组。

- [batch_size, output_size]
- [batch_size, hidden_size]

同时，返回具有如下形状的两个数组。

- [batch_size, feature_size]
- [batch_size, hidden_size]

这就是 RNNLayer 类的工作原理。现在只剩下描述 RNNNode 类了，它是 RNN 的核心，也是实际执行计算的地方。在开始描述之前，先弄清楚 RNNNode 类及其变体在整个 RNN 中的作用。

6.5.2 RNNNode类的基本元素

在大多数情况下，学习 RNN 会从讨论 RNNNode 类的工作方式开始。但是，要想理解 RNN，相比这一部分，前面在示意图和代码中所介绍的内容更为重要，包括数据的结构以及数据和隐藏状态在层和时间之间的路由方式。因此，本章直到现在才来讨论 RNNNode 类的工作方式。事实证明，可以采用多种方法来实现 RNNNode 类，基于给定时间步长实际处理数据，以及更新层的隐藏状态。还有一种方法可以生成“常规”[6] 的 RNN，这种 RNN 常被称作 vanilla RNN，下文会使用这个术语。然而，还有其他更复杂的方法可以生成不同的 RNN。例如，其中有一种变体叫作 GRU，代表“门控循环单元”（Gated Recurrent Unit）。通常，GRU 和其他 RNN 变体被描述成与 vanilla RNN 截然不同的对象。但是，所有 RNN 变体都共享目前所见的层的结构，了解这一点很重要。例如，它们都以相同的方式及时向前传递数据，在每个时间步长中更新它们的隐藏状态，唯一的区别在于这些“节点”具有不同的内部工作方式。

强调一点：如果实现 GRULayer 类而不是 RNNLayer 类，则代码将完全相同！以下代码仍将构成前向传递的核心：

```
sequence_length = x_seq_in.shape[1]

x_seq_out = np.zeros((batch_size, sequence_length, self.output_size))

for t in range(sequence_length):

    x_in = x_seq_in[:, t, :]

    y_out, H_in = self.nodes[t].forward(x_in, H_in, self.params)

    x_seq_out[:, t, :] = y_out
```

唯一的变化就是 self.nodes 中的每个“节点”都是 GRUNode 类，而不是 RNNNode 类。同理，backward 方法也是相同的。

对 vanilla RNN 最常见的变体 LSTM[7] 单元来说，以上描述几乎是完全正确的。唯一的区别是，LSTMLayer 要求该层“记住”**两个**量，并在序列元素随时间向前传递时进行更新：除了“隐藏状态”，该层还存储了“单元状态”，这样可以更好地对长期依赖关系进行建模。因此，在实现方式上，LSTMLayer 类相对于 RNNLayer 类存在一些细微的差异。例如，LSTMLayer 类在整个时间步长中将有两个 ndarray 来存储层的状态，具体如下。

- 一个 ndarray 是 start_H，形状为 [1, hidden_size]，表示层的隐藏状态。
- 一个 ndarray 是 start_C，形状为 [1, cell_size]，表示层的单元状态。

注 6：只是一种约定俗成的常见用法，没有具体标准。

注 7：Long Short-Term Memory，长短期记忆。

因此，每个 LSTMNode 类都应接受输入、隐藏状态以及单元状态。在前向传递时，代码如下所示：

```
y_out, H_in, C_in = self.nodes[t].forward(x_in, H_in, C_in self.params)
```

在后向传递时，代码如下所示：

```
grad_out, h_in_grad, c_in_grad = \
    self.nodes[t].backward(x_out_grad, h_in_grad, c_in_grad, self.params)
```

除了上面提到的三种情况，RNN 还有其他许多变体，其中一些除了隐藏状态外还具有单元状态，比如带有“窥孔连接”（peephole connection）的 LSTM；另一些则仅维持一个隐藏状态。与前面所介绍的变体一样，由 LSTMPeepholeConnectionNode 组成的层将以相同的方式输入 RNNLayer 类，因此具有相同的 forward 方法和 backward 方法。RNN 的基本结构包括在前向传递过程中数据在各层之间和各时间步长之间的路由方式，以及在后向传递过程中沿相反方向路由的方式，正是这样的结构让 RNN 显得独特。尽管 vanilla RNN 和基于 LSTM 的 RNN 在性能上有显著差异，但它们之间的结构差异实际上相对较小。

接下来看一下 RNNNode 类的实现。

6.5.3 vanilla RNNNode类

RNN 一次接收一个序列元素的数据。如果要预测石油价格，则在每个时间步长中，RNN 都会收到有关该时间步长用于预测价格的特征信息。另外，RNN 在其“隐藏状态”中持有一个编码，该编码表示在先前时间步长中所发生的事情的累积信息。我们希望组合这两种数据，即时间步长的特征以及所有先前时间步长的累积信息，使之成为该时间步长的预测以及更新后的隐藏状态。

要了解 RNN 如何完成这个任务，可以回顾常规神经网络中的情况。在前馈神经网络中，每一层都从前一层接收一组“已学习的特征”，每一个特征都是神经网络已“学习”的有用的原始特征的组合。然后，该层将这些特征与权重矩阵相乘，其中，权重矩阵可以让该层学习特征，而这些特征是该层接收的作为输入的特征的组合。为了分别对输出进行层级设置和归一化，可以向这些新特征添加一个偏差项，并将其输入到一个激活函数中。

在 RNN 中，我们希望更新后的隐藏状态是输入和旧隐藏状态的组合。因此，与常规神经网络类似，执行下面两个操作。

1. 将输入和隐藏状态连接起来。然后，将这个值乘以一个权重矩阵，加上一个偏差项，接着将结果输入到 Tanh 激活函数中。这便是更新后的隐藏状态。
2. 用权重矩阵乘以这个新的隐藏状态，该矩阵将隐藏状态转换为具有所需维度的输出。例如，在每个时间步长中，如果都使用这个 RNN 预测单个连续值，则将隐藏状态乘以一个大小为 [hidden_size, 1] 的权重矩阵。

因此，更新后的隐藏状态将是一个函数，该函数涉及在该时间步长中接收的输入以及先前的隐藏状态，而输出则是把这个更新后的隐藏状态输入给全连接层进行运算后所得到的结果。接下来对这个过程进行编码。

1. RNNNode 类：前向传递

下面的代码实现了上面描述的步骤。注意，就像稍后要处理的 GRU 和 LSTM 一样（以及第 1 章中针对简单数学函数所执行的操作），这里将前向传递中计算出的所有量都作为属性存储在 RNNNode 类中，这样可以使用它们来计算后向传递：

```
def forward(self,
            x_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]
            ) -> Tuple[ndarray]:
    '''
    param x：形状为[batch_size, vocab_size]的numpy数组。
    param H_prev：形状为[batch_size, hidden_size]的numpy数组。
    return self.x_out：形状为[batch_size, vocab_size]的numpy数组。
    return self.H：形状为[batch_size, hidden_size]的numpy数组。
    '''
    self.X_in = x_in
    self.H_in = H_in

    self.Z = np.column_stack((x_in, H_in))

    self.H_int = np.dot(self.Z, params_dict['W_f']['value']) \
                            + params_dict['B_f']['value']

    self.H_out = tanh(self.H_int)

    self.X_out = np.dot(self.H_out, params_dict['W_v']['value']) \
                                + params_dict['B_v']['value']

    return self.X_out, self.H_out
```

还需要注意一点：由于这里没有使用 ParamOperation，因此需要以其他方式存储参数。我们将参数存储在 params_dict 字典中，该字典按名称引用参数。此外，每个参数都有两个键：value 和 deriv，分别存储实际参数值及其关联的梯度。在本例的前向传递中，仅使用 value 键。

2. RNNNode 类：后向传递

假设给定损失相对于 RNNNode 类输出的梯度，那么 RNNNode 类的后向传递仅计算损失相对于 RNNNode 类输入的梯度值。可以使用类似第 1 章和第 2 章讨论的逻辑来执行此过程，因为可以将 RNNNode 类表示为一系列运算，所以可以简单地计算每个运算在其输入处的导数，然后将这些导数依次与之前的导数相乘（注意正确处理矩阵乘法），最后得到一个 ndarray，表示每个输入的损失梯度。以下代码实现了这一点：

```python
    def backward(self,
                 X_out_grad: ndarray,
                 H_out_grad: ndarray,
                 params_dict: Dict[str, Dict[str, ndarray]]) -> Tuple[ndarray]:
        '''
        param x_out_grad: numpy array of shape (batch_size, vocab_size)
        param h_out_grad: numpy array of shape (batch_size, hidden_size)
        param RNN_Params: RNN_Params object
        return x_in_grad: numpy array of shape (batch_size, vocab_size)
        return h_in_grad: numpy array of shape (batch_size, hidden_size)
        '''
        param x_out_grad：形状为[batch_size, vocab_size]的numpy数组。
        param h_out_grad：形状为[batch_size, hidden_size]的numpy数组。
        param RNN_Params：RNN_Params对象。
        return x_in_grad：形状为[batch_size, vocab_size]的numpy数组。
        return h_in_grad：形状为[batch_size, hidden_size]的numpy数组。
        '''
        assert_same_shape(X_out_grad, self.X_out)
        assert_same_shape(H_out_grad, self.H_out)

        params_dict['B_v']['deriv'] += X_out_grad.sum(axis=0)
        params_dict['W_v']['deriv'] += np.dot(self.H_out.T, X_out_grad)

        dh = np.dot(X_out_grad, params_dict['W_v']['value'].T)
        dh += H_out_grad

        dH_int = dh * dtanh(self.H_int)

        params_dict['B_f']['deriv'] += dH_int.sum(axis=0)
        params_dict['W_f']['deriv'] += np.dot(self.Z.T, dH_int)

        dz = np.dot(dH_int, params_dict['W_f']['value'].T)

        X_in_grad = dz[:, :self.X_in.shape[1]]
        H_in_grad = dz[:, self.X_in.shape[1]:]

        assert_same_shape(X_out_grad, self.X_out)
        assert_same_shape(H_out_grad, self.H_out)

        return X_in_grad, H_in_grad
```

注意，就像之前的 Operation 类一样，backward 方法的输入形状必须与 forward 方法的输出形状相匹配，backward 方法的输出形状则必须与 forward 方法的输入形状相匹配。

6.5.4 vanilla RNNNode类的局限性

注意，RNN 的目的是为数据序列中的依赖关系建模。以对石油价格建模为例，这意味着应该能够揭示在过去几个时间步长中看到的特征序列与下一个时间步长中油价变化之间的关系。但是，这里的“几个时间步长”应该是多长时间呢？可以想象一下，以石油价格为例，对于预测明天的石油价格，昨天的石油价格（前一个时间步长）要比前天的石油价格更为重要，而且时间越靠前，重要性往往越小。

尽管上面这种情况在许多实际问题中成立，但是在某些领域中还是可以应用RNN，而且在这个过程中可以学习超长期的依赖关系。一个典型的例子就是**语言建模**（language modeling），即在非常长的一系列现有单词或字符的基础上，构建一个可以预测下一个字符、单词或单词段的模型。这种应用程序非常普遍，本章稍后将讨论一些特定于语言建模的细节。vanilla RNN 通常不能完全处理这种情况。由于我们在前面已经了解了关于它的详细信息，因此可以理解其中的原因：在每个时间步长中，隐藏状态都将乘以该层中所有时间步长的**同一权重矩阵**。假如一次又一次地把一个数字乘以一个值 x，这时会发生什么？不难想象，如果 $x<1$，则数字呈指数下降到 0；如果 $x>1$，则数字呈指数增长到正无穷。RNN 具有相同的问题：在较长的时间范围内，由于同一组权重在每个时间步长中都乘以隐藏状态，因此这些权重的梯度会变得非常小或非常大。前者称为**梯度消失问题**（vanishing gradient problem），后者称为**梯度爆炸问题**（exploding gradient problem）。无论是哪种情况，对于训练 RNN 来实现高质量语言建模所需的超长期的依赖关系（50 ~ 100 个时间步长），两者都让这个过程更加困难。接下来将介绍对 vanilla RNN 架构的两种常用改进，这两种改进方法都可以大大缓解上述问题。

6.5.5 GRUNode类

vanilla RNN 将输入和隐藏状态组合起来，利用矩阵乘法来确定如何将隐藏状态中的信息与新输入中的信息进行“加权”，进而预测输出。这里使用更高级的 RNN 变体，主要是考虑到为了建模长期依赖关系（例如语言中存在的依赖关系），我们有时候会收到信息，这些信息指出了需要“忘记”或“重置”的隐藏状态。举一个简单的例子，比如句点“.”或冒号“:”，如果语言模型收到其中一个，它就知道应该忘记之前出现的字符，并开始按字符序列建立新的模式。

vanilla RNN 的第一个简单变体是 GRU，即**门控循环单元**，之所以这样命名，是因为输入和先前的隐藏状态是通过一系列“门”进行传递的。

(1) 第一个门类似于在 vanilla RNN 中发生的运算：将输入状态和隐藏状态连接在一起，乘以一个权重矩阵，然后传递给一个 sigmoid 运算。可以将它的输出看作“更新”门。
(2) 第二个门可以理解为“复位”门：将输入和隐藏状态连接起来，乘以一个权重矩阵，执行 sigmoid 运算，然后再乘以先前的隐藏状态。在给定输入的情况下，可以让神经网络“学习忘记”隐藏状态中的内容。
(3) 让第二个门的输出乘以另一个矩阵，并执行 Tanh 函数，输出是新隐藏状态的一个“候选状态”。
(4) 将隐藏状态更新为更新门并乘以新隐藏状态的“候选状态”，再加上旧隐藏状态与“1减去更新门”的乘积。



本章将介绍 vanilla RNN 的两个高级变体：GRU 和 LSTM。LSTM 更受欢迎，也比 GRU 出现得早。不管怎样，GRU 是 LSTM 的简单版本，并且更直接地说明了“门”这一概念如何使 RNN 在收到输入后能够“学习重置”隐藏状态，这就是首先介绍它的原因。

1. GRUNode 类：示意图

图 6-8 将 GRUNode 类描绘为一系列门。每个门都包含 Dense 层的运算：与权重矩阵相乘，加上偏差项并将结果输入给一个激活函数。所使用的激活函数可以是 sigmoid 函数，这种情况下结果的范围是 0 ～ 1；也可以是 Tanh 函数，这种情况下范围是 –1 ～ 1。接下来生成的每个中间 ndarray 的范围都会在数组名称下显示。

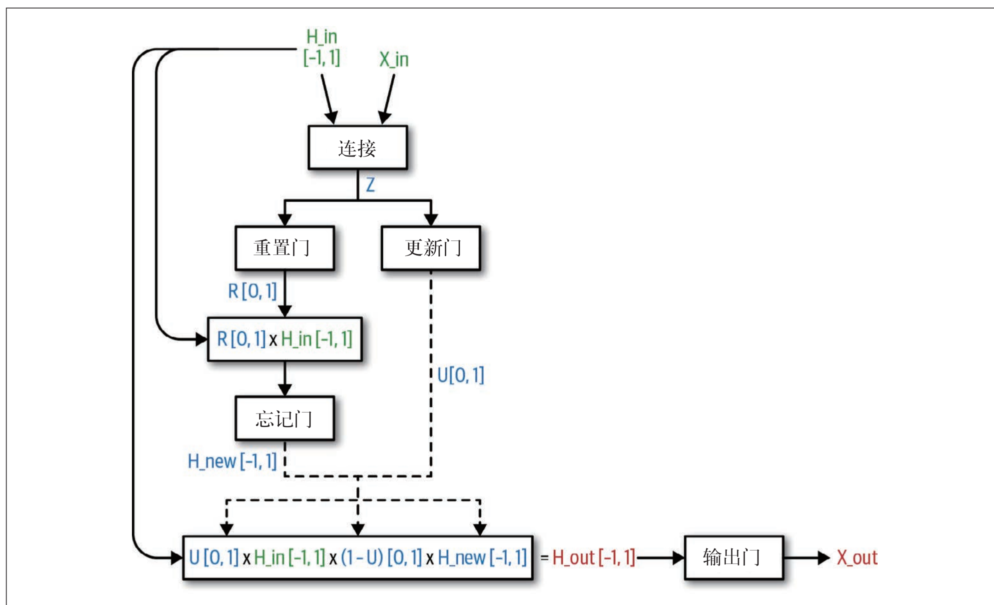


图 6-8：数据流向前输入到一个 GRUNode 类中，通过门并生成 X_out 和 H_out

在图 6-8、图 6-9 和图 6-10 中，节点中带有 in 后缀的为输入，带有 out 后缀的为输出，其余为计算出的中间量。所有权重（未直接显示）都包含在门中。

注意，要对此进行反向传播，只能将其表示为一系列 Operation 类，计算每个 Operation 类相对于其输入的导数，然后将结果相乘。这里没有明确展示这一点，而是将门（实际上是 3 个运算）显示为一个块。尽管如此，在这一点上，我们仍然知道如何通过构成每个门的 Operation 类进行反向传播，并且由于“门”的概念贯穿于整个对 RNN 及其变体的描述中，因此这里坚持使用这种表示方法。

实际上，图 6-9 显示了 vanilla RNNNode 类的一种表示方式，其中使用了“门”的概念。

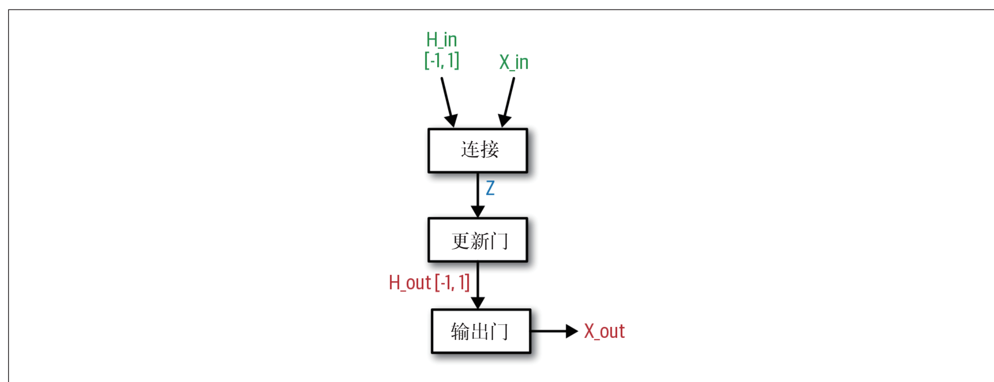


图 6-9：数据流向前输入到一个 RNNNode 类中，仅通过两个门并生成 X_out 和 H_out

可见，针对之前描述的构成 vanilla RNNNode 类的 Operation 类，另一种解决方法是将输入和隐藏状态传递给两个门。

2. GRUNode 类：代码

以下代码为前面所述的 GRUNode 类实现了前向传递：

```
def forward(self,
            X_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]) -> Tuple[ndarray]:
    '''
    param X_in：形状为[batch_size, vocab_size]的numpy数组。
    param H_in：形状为[batch_size, hidden_size]的numpy数组。
    return self.X_out：形状为[batch_size, vocab_size]的numpy数组。
    return self.H_out：形状为[batch_size, hidden_size]的numpy数组。
    '''
    self.X_in = X_in
    self.H_in = H_in

    # 重置门
    self.X_r = np.dot(X_in, params_dict['W_xr']['value'])
    self.H_r = np.dot(H_in, params_dict['W_hr']['value'])

    # 更新门
    self.X_u = np.dot(X_in, params_dict['W_xu']['value'])
    self.H_u = np.dot(H_in, params_dict['W_hu']['value'])

    # 门
    self.r_int = self.X_r + self.H_r + params_dict['B_r']['value']
    self.r = sigmoid(self.r_int)
    self.u_int = self.X_r + self.H_r + params_dict['B_u']['value']
    self.u = sigmoid(self.u_int)

    # 新状态
    self.h_reset = self.r * H_in
    self.X_h = np.dot(X_in, params_dict['W_xh']['value'])
```

```
        self.H_h = np.dot(self.h_reset, params_dict['W_hh']['value'])
        self.h_bar_int = self.X_h + self.H_h + params_dict['B_h']['value']
        self.h_bar = np.tanh(self.h_bar_int)

        self.H_out = self.u * self.H_in + (1 - self.u) * self.h_bar

        self.X_out = (
            np.dot(self.H_out, params_dict['W_v']['value']) \
            + params_dict['B_v']['value']
            )

        return self.X_out, self.H_out
```

注意，这里没有明确地将 `X_in` 和 `H_in` 连接起来，与 `RNNNode` 类将它们一起使用不同，`GRUNode` 类会独立使用它们。具体来说，`self.h_reset = self.r * H_in` 只使用了 `H_in`，而没有使用 `X_in`。

可以在本书的随书文件中找到 `backward` 方法[8]。它只是向后传递到那些组成 `GRUNode` 类的运算，计算每个运算相对于其输入的导数，并将结果相乘。

6.5.6 LSTMNode类

LSTM 是 vanilla RNN 最流行的变体。部分原因是，它诞生于深度学习早期，可以追溯到 1997 年[9]。直到近几年，对 LSTM 替代品的研究才获得进展，例如，GRU 是在 2014 年提出的。

和 GRU 一样，提出 LSTM 的动机是希望让 RNN 在接收新输入时能够“重置”或“忘记”其隐藏状态。在 GRU 中，这是通过一系列门提供输入和隐藏状态来实现的，并且可以使用这些门计算新隐藏状态（比如使用 `self.r` 门计算 `self.h_bar`），然后使用新隐藏状态和旧隐藏状态的加权平均值来计算最终隐藏状态，这个过程由更新门进行控制：

```
self.H_out = self.u * self.H_in + (1 - self.u) * self.h_bar
```

相反，**LSTM 使用单独的“状态”向量（“单元状态”）来判断是否“忘记”隐藏状态中的内容**。然后，使用其他两个门来控制应重置或更新**单元状态中内容**的程度，并使用第四个门根据最终单元状态来确定隐藏状态更新的程度[10]。

1. LSTMNode 类：示意图

图 6-10 展示了 `LSTMNode` 类的示意图，其中的运算表示为门。

注 8：请从图灵社区下载随书文件：ituring.cn/book/2759。——编者注

注 9：参见 Sepp Hochreiter 等人于 1997 年发表的 LSTM 原始论文“Long Short-Term Memory”。

注 10：至少 LSTM 的标准变体遵循这个规则。前面提到还有其他变体，例如“带窥孔连接的 LSTM”，其中，门的布置方式有所不同。

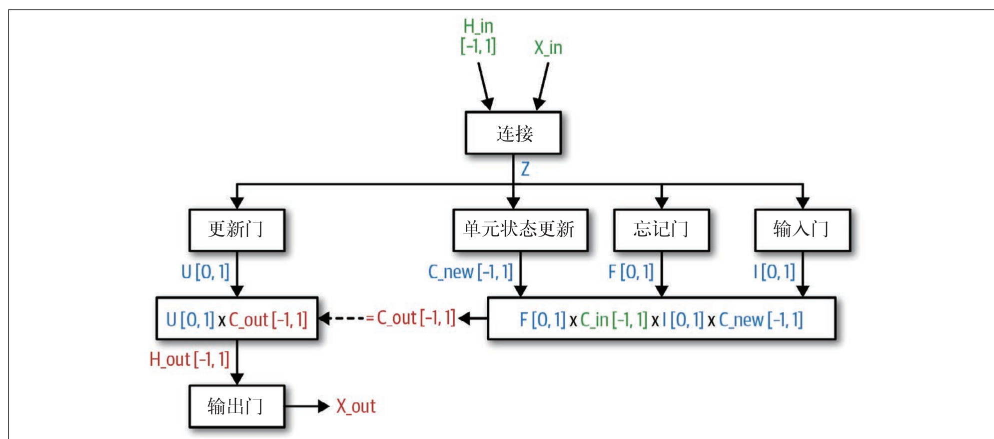


图 6-10：数据流向前输入到一个 LSTMNode 类中，经过一系列门并分别输出更新后的单元状态 C_out、隐藏状态 H_out 以及实际的输出 X_out

2. LSTMNode 类：代码

与 GRUNode 类一样，本书的随书文件也提供了 LSTMNode 类的完整代码，包括 backward 方法以及展示节点如何适应 LSTMLayer 类的示例[11]。这里仅展示 forward 方法：

```
def forward(self,
            X_in: ndarray,
            H_in: ndarray,
            C_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]):
    '''
    param X_in：形状为[batch_size, vocab_size]的numpy数组。
    param H_in：形状为[batch_size, hidden_size]的numpy数组。
    param C_in：形状为[batch_size, hidden_size]的numpy数组。
    return self.x_out：形状为[batch_size, output_size]的numpy数组。
    return self.H：形状为[batch_size, hidden_size]的numpy数组。
    return self.C：形状为[batch_size, hidden_size]的numpy数组。
    '''

    self.X_in = X_in
    self.C_in = C_in

    self.Z = np.column_stack((X_in, H_in))
    self.f_int = (
        np.dot(self.Z, params_dict['W_f']['value']) \
        + params_dict['B_f']['value']
        )
    self.f = sigmoid(self.f_int)

    self.i_int = (
        np.dot(self.Z, params_dict['W_i']['value']) \
        + params_dict['B_i']['value']
        )
```

注 11：请从图灵社区下载随书文件：ituring.cn/book/2759。——编者注

```
        self.i = sigmoid(self.i_int)

        self.C_bar_int = (
            np.dot(self.Z, params_dict['W_c']['value']) \
            + params_dict['B_c']['value']
            )
        self.C_bar = tanh(self.C_bar_int)
        self.C_out = self.f * C_in + self.i * self.C_bar

        self.o_int = (
            np.dot(self.Z, params_dict['W_o']['value']) \
            + params_dict['B_o']['value']
            )
        self.o = sigmoid(self.o_int)
        self.H_out = self.o * tanh(self.C_out)

        self.X_out = (
            np.dot(self.H_out, params_dict['W_v']['value']) \
            + params_dict['B_v']['value']
            )

        return self.X_out, self.H_out, self.C_out
```

这是 RNN 框架中启动模型训练所需的最后一个元素！还有一个应该讨论的主题：如果要将文本数据输入到 RNN 中，那么应该用哪种形式来表示文本数据呢？

6.5.7 基于字符级RNN语言模型的数据表示

语言建模是 RNN 最常见的任务之一。如何将字符序列重塑为训练数据集，从而训练 RNN 来预测下一个字符？最简单的方法是使用**独热编码**（one-hot encoding），下面介绍它的工作方式。首先，每个字母都表示为一个向量，其维数等于**词汇量**，或者等于将接受神经网络训练的整个文本语料库中的字母数（该值预先计算并硬编码为神经网络中的超参数）。然后，将向量中对应字母位置上的值设为 1，其他位置上的值设为 0。最后，将每个字母的向量简单地连接在一起，获得字母序列的整体表示。

这里有一个简单的示例，其中展示了一个包含字母 *a*、*b*、*c* 和 *d* 的词汇表，我们将 *a* 称为第一个字母，*b* 称为第二个字母，以此类推 [12]：

$$abcdb \rightarrow \left[\begin{bmatrix}1\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\right] = \begin{bmatrix}10000\\01001\\00100\\00010\end{bmatrix}$$

这个二维数组将代替一批序列中的形状为 `(sequence_length, num_features) = (5, 4)` 的观测值。也就是说，如果文本是 *abcdba*（长度为 6），并且需要在数组中输入长度为 5 的序列，则第一个序列将转换为以上矩阵，第二个序列则将转换为以下矩阵：

注 12：这个顺序是任意的，可以随机定义。

$$bcdba \rightarrow \begin{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 00001 \\ 10010 \\ 01000 \\ 00100 \end{bmatrix}$$

然后将它们串联在一起，创建一个形状为 `[batch_size, sequence_length, vocab_size] = (2, 5, 4)` 的 RNN 输入。继续使用这种方法，可以获取原始文本并将其转换为一批序列，然后输入到 RNN 中。

在本书的随书文件中，我把这种方法作为可以接收原始文本的 `RNNTrainer` 类的一部分进行编码，同时使用这里介绍的技术对其进行预处理，然后分批将其输入到 RNN 中。

6.5.8 其他语言建模任务

虽然本章在前面没有强调这一点，但是从前面的代码可以看出，在所有 `RNNNode` 类的变体中，`RNNLayer` 类所输出的特征数量都可以与输入的不同。所有变体的最后一步都是将神经网络的最终隐藏状态乘以通过 `params_dict[W_v]` 获取的权重矩阵。权重矩阵的第二个维度将确定 `Layer` 类输出的维数，这样一来，只需更改每个 `Layer` 类中的 `output_size` 参数，就可以将相同的架构用于不同的语言建模任务。

比方说，目前我们只考虑通过“预测下一个字符”来构建一个语言模型。在这种情况下，输出大小将等于词汇表的大小（词汇量），即 `output_size = vocab_size`。但是，对于情感分析之类的内容，传入的序列可能只是带有“0”或“1”的标签（表示正或负）。这时不仅 `output_size = 1`，而且只有在整个序列传递完成之后，才将输出与目标进行比较，如图 6-11 所示。

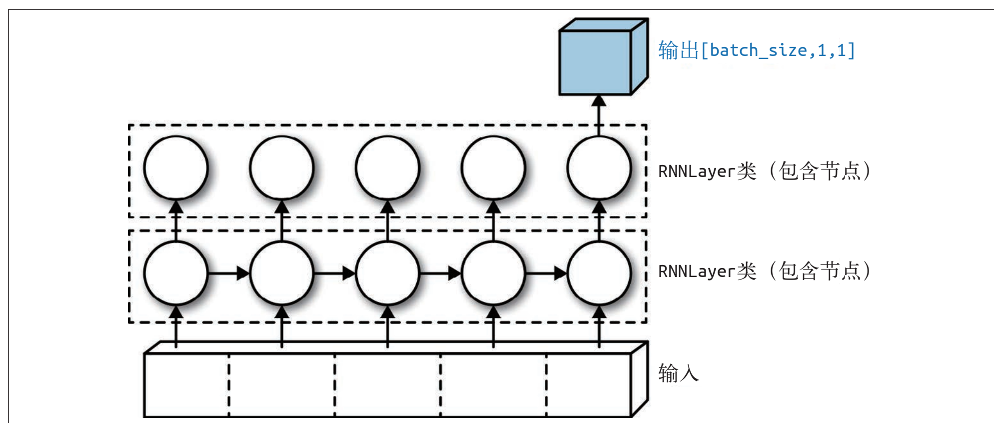


图 6-11：对于情感分析，RNN 将预测值与实际值进行比较，并仅针对最后一个序列元素的输出产生梯度。反向传播将照常进行，每个节点（不包括最后一个节点）仅接收一个所有元素都为 0 的 `X_grad_out` 数组

因此，这个框架可以很容易地适应不同的语言建模任务。事实上，它可以适应所有针对连续性数据的建模任务，并且可以一次将一个序列元素输入到神经网络中。

在总结本节之前，先来看关于 RNN 的一个不常讨论的方面：可以对不同种类的层（GRULayer 类、LSTMLayer 类和其他变体）进行混合和匹配。

6.5.9　组合RNNLayer类的变体

堆叠不同种类的 RNNLayer 类很简单：每个 RNN 都会输出一个形状为 [batch_size, sequence_length, output_size] 的 ndarray，可以将其传递到下一层。与 Dense 层一样，不必指定 input_shape。在给定输入的情况下，只需根据层接收到的第一个 ndarray 输入将权重设置为合适的形状。因此，RNNModel 可以具有以下 self.layers 属性：

```
[RNNLayer(hidden_size=256, output_size=128),
RNNLayer(hidden_size=256, output_size=62)]
```

与全连接神经网络一样，只需确保最后一层生成所需维数的输出即可。这就像在处理 MNIST 问题的全连接神经网络中，最后一层的维数必须为 10，如果这里要处理一个大小为 62 的词汇表并预测下一个字符，那么最后一层的 output_size 参数必须为 62。

在学习完本章之后，有一点应该很清楚：因为前文介绍的每种层都具有相同的基础结构，其中包含 feature_size 维的输入序列和 output_size 维的输出序列，所以可以轻松地堆叠不同种类的层。例如，可以训练一个具有以下 self.layers 属性的 RNNModel：

```
[GRULayer(hidden_size=256, output_size=128),
LSTMLayer(hidden_size=256, output_size=62)]
```

换句话说，第一层使用 GRUNode 类向前传递输入，然后将形状为 [batch_size, sequence_length, 128] 的 ndarray 传递到下一层，下一层随后将其传递给该层的 LSTMNode 类。

6.5.10　将全部内容整合在一起

展示 RNN 有效性的经典方法就是训练模型用特定的风格书写文本。本书的随书文件提供了完整的模型示例，该模型使用本章介绍的抽象定义，可以学习以莎士比亚的风格书写文本。目前唯一还没有展示的组件是 RNNTrainer 类，该类遍历训练数据，对其进行预处理并传递给模型。这与之前看到的 Trainer 类的主要区别在于，对 RNN 来说，一旦选择了要输入的一批数据（批次中的每个元素都只是一个字符串），就必须首先对其进行预处理，即对每个字母进行一次独热编码，并将得到的向量连接成一个序列，从而将每个长度为 sequence_length 的字符串转换为形状为 [sequence_length, vocab_size] 的 ndarray。为了形成要输入到 RNN 的批次，这些 ndarray 将连接在一起，形成大小为 [sequence_length, vocab_size, batch_size] 的批数据。

但是，一旦对数据进行了预处理并定义了模型，就可以像前面介绍的其他神经网络一样训练 RNN：迭代输入批次，通过比较模型的预测与目标来生成损失，然后反向传播损失，从而更新权重。

6.6　小结

本章介绍了 RNN，这是一种特殊的神经网络架构，旨在处理数据序列而不是单个运算。在 RNN 中，各层向前传递数据，并随着数据的传递更新其隐藏状态（在 LSTM 中，更新它们的单元状态）。本章还介绍了 RNN 的高级变体（GRU 和 LSTM），以及它们如何在每个时间步长中通过一系列“门”向前传递数据。但是，由于这些高级变体在本质上以相同的方式处理数据序列，因此它们的层具有相同的结构，只是在每个时间步长中应用的特定运算有所不同。

通过本章的介绍，你应该对 RNN 有了一定的了解。第 7 章将对本书内容进行总结，通过将深度学习技术付诸实践，来展示如何使用 PyTorch 框架实现前面介绍的所有内容。PyTorch 框架是一款基于自动微分的高性能框架，用于构建和训练深度学习模型。加油！

第 7 章

PyTorch

第 5 章和第 6 章分别从零开始实现了 CNN 和 RNN。不过，尽管有必要了解它们的工作原理，但仅凭这些知识并不能解决实际问题。为此，你需要通过一个高性能库来实现它们。当然，我们可以自己构建一个高性能的神经网络库，但是那将需要一整本书的篇幅来描述（篇幅只会更长）。相反，本章只介绍 PyTorch，这是一款基于自动微分且越来越流行的神经网络框架。

本章继续遵循前文的讲述风格，在编写代码时，整个过程与神经网络工作原理的思维模型、`Layer` 类和 `Trainer` 类等的编写方式相对应。也就是说，本章不会按照 PyTorch 的一般做法来编写代码，但是本书的随书文件提供了相关链接，可以在那里了解更多用 PyTorch 表达神经网络的信息[1]。在此之前，先来看一下 PyTorch 的核心数据类型 `Tensor`，该数据类型可以让 PyTorch 实现自动微分，从而清晰地表达神经网络的训练过程。

7.1 PyTorch Tensor

第 6 章展示了一个简单的 `NumberWithGrad` 类，它通过跟踪对它执行的运算来累积梯度。这意味着如果按照下面这样编写，那么 `a.grad` 将等于 35，它实际上就是 `e` 相对于 `a` 的偏导数。

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = (a + 2)
```

注 1：请从图灵社区下载随书文件：ituring.cn/book/2759。——编者注

```
e = c * d
e.backward()
```

PyTorch 的 Tensor 类相当于 ndarrayWithGrad：它与 NumberWithGrad 类似，但使用数组（如 numpy）而不是仅使用 float 和 int。现在使用 Tensor 重写前面的示例。首先，手动初始化 Tensor：

```
a = torch.Tensor([[3., 3.,],
                  [3., 3.]], requires_grad=True)
```

注意下面两点。

- 就像处理 ndarray 一样，可以通过将其中包含的数据简单地包装在 torch.Tensor 中来初始化 Tensor。
- 当用这种方式初始化 Tensor 时，必须传递 requires_grad = True 参数，来告诉 Tensor 累积梯度。

当完成此操作后，可以像以前一样执行计算：

```
b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e_sum = e.sum()
e_sum.backward()
```

可以看到，与 NumberWithGrad 示例相比，这里还有一个额外的步骤：在调用 backward 方法之前，必须先对 e 进行**求和**。这是因为，正如第 1 章所论证的，考虑“一个数相对于一个数组的导数”是没有意义的。但是，可以计算 e_sum 相对于数组 a 中每个元素的偏导数。事实上，可以看到结果与前几章是一致的：

```
print(a.grad)

tensor([[35., 35.],
        [35., 35.]], dtype=torch.float64)
```

PyTorch 的这一特性可以用来定义模型，即通过定义前向传递、计算损失并在损失上调用 backward 方法，来自动计算每个 parameter 相对于该损失的导数。特别注意，这里不必担心前向传递中多次重用相同的量（回顾前几章使用的 Operation 框架的局限性）。正如这个简单的示例所示，一旦对计算的输出调用 backward 方法，就可以自动正确计算梯度。

接下来的几节将展示如何使用 PyTorch 的数据类型实现本书前面所介绍的训练框架。

7.2　使用PyTorch进行深度学习

如前所述，深度学习模型包含多个元素，这些元素通过协同工作来生成训练模型。

- 一个 Model 类，其中包含 Layer 类
- 一个 Optimizer 类
- 一个 Loss 类
- 一个 Trainer 类

事实证明，使用 PyTorch 来实现 Optimizer 类和 Loss 类都只需编写一行代码，Model 类和 Layer 类实现起来也很简单，接下来将依次介绍这些元素。

7.2.1 PyTorch元素：Model类及其Layer类

PyTorch 的一个关键特性是能够将模型和层定义为易于使用的对象，这些对象仅需从 torch.nn.Module 类中继承，用于处理向后发送梯度并自动存储参数。本章稍后将介绍如何把这些部分组合在一起。现在，只需知道 PyTorchLayer 类可以写成如下形式：

```
from torch import nn, Tensor

class PyTorchLayer(nn.Module):

    def __init__(self) -> None:
        super().__init__()

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        raise NotImplementedError()
```

PyTorchModel 类可以写成如下形式：

```
class PyTorchModel(nn.Module):

    def __init__(self) -> None:
        super().__init__()

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        raise NotImplementedError()
```

换句话说，PyTorchLayer 类或 PyTorchModel 类的每个子类都只需要实现 _init_ 方法和 forward 方法，以便直接使用它们[2]。

inference 标志位

如第 4 章所述，由于 dropout 的应用，我们需要适时改变模型的行为，具体要取决于它是在训练模式下运行还是在推理模式下运行。在 PyTorch 中，可以通过在模型或层（或任何从 nn.Module 类中继承的对象）上运行 m.eval，从而将模型或层从默认的训练模式切换到

注 2：用这种方式编写 Layer 类和 Model 类不是 PyTorch 的最常见或推荐的用法，之所以在这里展示这种方式，是因为它与目前介绍的概念更接近。如果使用 PyTorch 来构建神经网络基本要素，那么可以浏览 PyTorch 官网，其中有更为常见的方法。

推理模式。此外，PyTorch 提供了一种简单的方法，可以使用 apply 函数快速更改层的所有子类的行为。如果像下面这样定义：

```
def inference_mode(m: nn.Module):
    m.eval()
```

那么可以在定义的 PyTorchModel 类或 PyTorchLayer 类的每个子类的 forward 方法中都包含如下代码，从而获得预期的标志位。

```
if inference:
    self.apply(inference_mode)
```

接下来看一下如何把它们整合在一起。

7.2.2 使用PyTorch实现神经网络基本要素：DenseLayer类

现在已经具备了实现 Layer 类的所有先决条件，但是需要借助 PyTorch 运算。DenseLayer 类的编写方式如下：

```
class DenseLayer(PyTorchLayer):
    def __init__(self,
                 input_size: int,
                 neurons: int,
                 dropout: float = 1.0,
                 activation: nn.Module = None) -> None:
        super().__init__()
        self.linear = nn.Linear(input_size, neurons)
        self.activation = activation
        if dropout < 1.0:
            self.dropout = nn.Dropout(1 - dropout)
            def forward(self, x: Tensor,
                        inference: bool = False) -> Tensor:
                if inference:
                    self.apply(inference_mode)

                x = self.linear(x) # does weight multiplication + bias
                if self.activation:
                    x = self.activation(x)
                if hasattr(self, "dropout"):
                    x = self.dropout(x)

                return x
```

基于 nn.Linear，我们看到了第一个 PyTorch 运算示例，该运算自动处理反向传播。这个对象不仅在前向传递中处理权重乘积和增加偏差项，还会累积 x 的梯度，从而在后向传递中正确计算出损失相对于参数的导数。还要注意，由于所有 PyTorch 运算都从 nn.Module 类中继承，因此可以像数学函数那样调用它们。例如，在前面的场景中，可以使用 self.linear(x)，而不是 self.linear.forward(x)。这对于 DenseLayer 类本身也是如此，我们将在下面的模型中使用它。

7.2.3 示例：基于PyTorch的波士顿房价模型

使用 `Layer` 类作为基本要素，可以实现第 2 章和第 3 章所介绍的房价模型。回想一下，这个模型只有一个带有 sigmoid 激活函数的隐藏层。第 3 章在面向对象的框架中实现了这一点，该框架具有一个 `Layer` 类和一个模型，其中模型包含一个长度为 2 的列表作为其 `layers` 属性。同理，可以定义 `HousePricesModel` 类，该类从 `PyTorchModel` 类中继承，如下所示：

```
class HousePricesModel(PyTorchModel):

    def __init__(self,
                 hidden_size: int = 13,
                 hidden_dropout: float = 1.0):
        super().__init__()
        self.dense1 = DenseLayer(13, hidden_size,
                                 activation=nn.Sigmoid(),
                                 dropout = hidden_dropout)
        self.dense2 = DenseLayer(hidden_size, 1)

    def forward(self, x: Tensor) -> Tensor:

        assert_dim(x, 2)

        assert x.shape[1] == 13

        x = self.dense1(x)
        return self.dense2(x)
```

然后可以通过以下方式对其进行实例化：

```
pytorch_boston_model = HousePricesModel(hidden_size=13)
```

注意，为 PyTorch 模型编写单独的 `Layer` 类不是常规做法。更常见的做法是根据正在运行的单个运算来定义模型，方法如下：

```
class HousePricesModel(PyTorchModel):

    def __init__(self,
                 hidden_size: int = 13):
        super().__init__()
        self.fc1 = nn.Linear(13, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, x: Tensor) -> Tensor:

        assert_dim(x, 2)

        assert x.shape[1] == 13

        x = self.fc1(x)
        x = torch.sigmoid(x)
        return self.fc2(x)
```

假如以后需要自行构建 PyTorch 模型，你可能希望以这种方式编写代码，而不是创建单独的 Layer 类。即使在阅读其他人的代码时，也总会看到与前面的代码相似的内容。

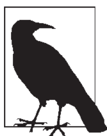
相比 Layer 类和 Model 类，Optimizer 类和 Loss 类要简单得多，下面来看一下具体介绍。

7.2.4 PyTorch元素：Optimizer类和Loss类

在 PyTorch 中，Optimizer 类和 Loss 类都可以通过一行代码来实现。例如，第 4 章介绍的 SGDMomentum 损失可以写成如下形式。

```
import torch.optim as optim

optimizer = optim.SGD(pytorch_boston_model.parameters(), lr=0.001)
```

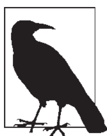


在 PyTorch 中，模型会作为参数传递给 Optimizer 类。这样可以确保优化器“指向”正确模型的参数，从而确定每次迭代要更新的内容（之前使用 Trainer 类完成了此操作）。

此外，第 2 章介绍的均方误差损失和第 4 章出现的 SoftmaxCrossEntropyLoss 可以简单写为：

```
mean_squared_error_loss = nn.MSELoss()
softmax_cross_entropy_loss = nn.CrossEntropyLoss()
```

正如前面的 Layer 类，它们从 nn.Module 类中继承，因此可以按照与 Layer 类相同的方式进行调用。



注意，nn.CrossEntropyLoss 类虽然在名称中没有出现 softmax，但的确对输入执行了 softmax 运算，因此就可以从神经网络传入“原始输出”，而不是像以前那样，传入已经通过 softmax 函数的输出。

与以前的 Layer 类一样，由于这些 Loss 类都从 nn.Module 类中继承，因此可以使用相同的方式进行调用，例如使用 loss(x)，而不是 loss.forward(x)。

7.2.5 PyTorch元素：Trainer类

Trainer 类将模型的所有元素融合在一起。它有什么具体要求呢？可以看到，它必须实现用于训练神经网络的通用模式，这在本书中多次出现。

- 将一批数据输入模型。
- 将输出和目标输入损失函数，计算损失值。
- 计算所有参数的损失梯度。
- 根据某些规则，使用 Optimizer 类更新参数。

在 PyTorch 中，这一切都以相同的方式实现，但还有两个小的注意事项。

- 在默认情况下，Optimizer 类将在每次参数更新迭代后保留参数的梯度（前面称为 param_grads 参数）。要在下一次参数更新之前清除这些梯度，可以调用 self.optim.zero_grad。
- 正如先前在简单自动微分示例中所看到的，要启动反向传播，必须在计算损失值后调用 loss.backward。

这导致在整个 PyTorch 训练循环中出现以下代码，实际上它们将在 PyTorchTrainer 类中使用。正如在前面章节的 Trainer 类中所做的那样，PyTorchTrainer 类将为一批数据（X_batch，y_batch）引入一个 Optimizer 类、一个 PyTorchModel 类和一个 Loss 类（nn.MSELoss 类或 nn.CrossEntropyLoss 类）。然后，把这些对象分别设置为 self.optim、self.model 和 self.loss。以下代码将训练模型：

```
# 将梯度设置为0
self.optim.zero_grad()

# 将X_batch输入模型
output = self.model(X_batch)

# 计算损失值
loss = self.loss(output, y_batch)

# 对损失执行后向调用，开始反向传播
loss.backward()

# 与以前一样，调用self.optim.step()来更新参数
self.optim.step()
```

以上是最重要的代码行。下面是 PyTorchTrainer 类的其余代码，其中大部分与前面章节中的 Trainer 类代码类似。

```
class PyTorchTrainer(object):
    def __init__(self,
                 model: PyTorchModel,
                 optim: Optimizer,
                 criterion: _Loss):
        self.model = model
        self.optim = optim
        self.loss = criterion
        self._check_optim_net_aligned()

    def _check_optim_net_aligned(self):
        assert self.optim.param_groups[0]['params']\
        == list(self.model.parameters())

    def _generate_batches(self,
                          X: Tensor,
                          y: Tensor,
                          size: int = 32) -> Tuple[Tensor]:
```

```
        N = X.shape[0]

        for ii in range(0, N, size):
            X_batch, y_batch = X[ii:ii+size], y[ii:ii+size]

            yield X_batch, y_batch


    def fit(self, X_train: Tensor, y_train: Tensor,
            X_test: Tensor, y_test: Tensor,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32):

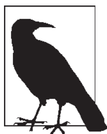
        for e in range(epochs):
            X_train, y_train = permute_data(X_train, y_train)

            batch_generator = self._generate_batches(X_train, y_train,
                                                     batch_size)

            for ii, (X_batch, y_batch) in enumerate(batch_generator):

                self.optim.zero_grad()
                output = self.model(X_batch)
                loss = self.loss(output, y_batch)
                loss.backward()
                self.optim.step()

            output = self.model(X_test)
            loss = self.loss(output, y_test)
            print(e, loss)
```



由于要向 `Trainer` 类传递 `Model` 类、`Optimizer` 类和 `Loss` 类，因此需要检查 `Optimizer` 类所引用的参数是否与模型的参数相同。`_check_optim_net_aligned` 执行此操作。

现在训练模型非常简单：

```
net = HousePricesModel()
optimizer = optim.SGD(net.parameters(), lr=0.001)
criterion = nn.MSELoss()

trainer = PyTorchTrainer(net, optimizer, criterion)

trainer.fit(X_train, y_train, X_test, y_test,
            epochs=10,
            eval_every=1)
```

相比第 1 ～ 3 章所构建的框架，在训练模型时，上面的代码与基于该框架的代码几乎相同。无论你是在底层使用 PyTorch、TensorFlow 还是 Theano，训练深度学习模型的要素都是一样的！

接下来将展示如何实现第 4 章介绍的优化技术，从而探索 PyTorch 的更多特性。

7.2.6　PyTorch优化学习技术

第 4 章介绍了以下 4 种优化技术。

- 动量
- dropout
- 权重初始化
- 学习率衰减

这些优化技术在 PyTorch 中都很容易实现。例如，要在优化器中包含动量，只需要在 `SGD` 中包含 `momentum` 关键字，这样优化器就变成下面这种形式：

```
optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

dropout 实现起来也同样容易。就像 PyTorch 中的内置模块 `nn.Linear(n_in, n_out)` 可以计算 `Dense` 层的运算一样，`nn.Dropout(dropout_prob)` 模块实现了 `Dropout` 运算，在默认情况下，传入的概率是丢弃给定神经元的概率，而不是像之前实现的保持神经元的概率。

无须担心权重初始化：对于大多数涉及参数的 PyTorch 运算（包括 `nn.Linear`），它们的权重会根据层的大小自动缩放。

另外，PyTorch 具有 `lr_scheduler` 类，该类可以在多轮训练后降低学习率，你所需要的关键导入项就来自 `torch.optim import lr_scheduler`[3]。对于这些从基本原理开始介绍的技术，现在可以轻松地将它们应用到接下来的深度学习项目中！

7.3　PyTorch中的CNN

第 5 章系统地介绍了 CNN 的工作原理，重点讨论了多通道卷积运算。可以看到，该运算将输入图像的像素转换为构成特征图的神经元层，其中每个神经元表示在图像中的该位置是否存在给定的视觉特征（由卷积过滤器定义）。多通道卷积运算的输入（两个）和输出具有以下形状。

- 输入数据形状为 `[batch_size, in_channels, image_height, image_width]`。
- 输入参数形状为 `[in_channels,  out_channels, filter_size,  filter_size]`。
- 输出形状为 `[batch_size, out_channels, image_height, image_width]`。

按照这种表示法，PyTorch 中的多通道卷积运算可以像下面这样表示：

```
nn.Conv2d(in_channels, out_channels, filter_size)
```

注 3：本书的随书文件提供了一个实现指数学习率衰减的代码示例，它是 `PyTorchTrainer` 的一部分。另外，PyTorch 网站提供了关于使用 `ExponentialLR` 类的文档。

使用这个定义，围绕该运算包装 `ConvLayer` 类就很简单。

```
class ConvLayer(PyTorchLayer):
    def __init__(self,
                 in_channels: int,
                 out_channels: int,
                 filter_size: int,
                 activation: nn.Module = None,
                 flatten: bool = False,
                 dropout: float = 1.0) -> None:
        super().__init__()

        # 该层的主要运算
        self.conv = nn.Conv2d(in_channels, out_channels, filter_size,
                              padding=filter_size // 2)

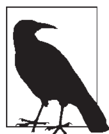
        # 与之前相同的“激活”运算和flatten运算
        self.activation = activation
        self.flatten = flatten
        if dropout < 1.0:
            self.dropout = nn.Dropout(1 - dropout)

    def forward(self, x: Tensor) -> Tensor:

        # 始终应用卷积运算
        x = self.conv(x)

        # 选择性地应用卷积运算
        if self.activation:
            x = self.activation(x)
        if self.flatten:
            x = x.view(x.shape[0], x.shape[1] * x.shape[2] * x.shape[3])
        if hasattr(self, "dropout"):
            x = self.dropout(x)

        return x
```



在第 5 章中，为了让输出图像的大小与输入图像的大小相同，我们根据过滤器的大小自动填充输出。基于同样的目标，在 PyTorch 中的做法则不同，我们在 `nn.Conv2d` 运算中添加了一个参数，设置 `padding = filter_size // 2`。

这样一来，只需要定义 `PyTorchModel` 即可开始训练。其中，`__init__` 方法包含该模型的相关运算，`forward` 方法定义了运算的序列。接下来是一个简单的架构，可以应用于第 4 章和第 5 章介绍的 MNIST 数据集，该架构包括以下两个方面。

- 两个卷积层：一个可以将输入从 1 个通道转换为 16 个通道；另一个可以将这 16 个通道转换为 8 个通道（每个通道仍包含 28×28 个神经元）。
- 两个全连接层。

在卷积架构中，对于几个卷积层后面紧跟着少量全连接层，这种模式很常见。这里只使用

两个全连接层：

```
class MNIST_ConvNet(PyTorchModel):
    def __init__(self):
        super().__init__()
        self.conv1 = ConvLayer(1, 16, 5, activation=nn.Tanh(),
                               dropout=0.8)
        self.conv2 = ConvLayer(16, 8, 5, activation=nn.Tanh(), flatten=True,
                               dropout=0.8)
        self.dense1 = DenseLayer(28 * 28 * 8, 32, activation=nn.Tanh(),
                                 dropout=0.8)
        self.dense2 = DenseLayer(32, 10)

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        x = self.conv1(x)
        x = self.conv2(x)

        x = self.dense1(x)
        x = self.dense2(x)
        return x
```

然后可以像训练 HousePricesModel 一样训练这个模型：

```
model = MNIST_ConvNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

trainer = PyTorchTrainer(model, optimizer, criterion)

trainer.fit(X_train, y_train,
            X_test, y_test,
            epochs=5,
            eval_every=1)
```

注意，对 nn.CrossEntropyLoss 类来说，还有一点很重要。回想一下，在前几章的自定义框架中，Loss 类期望输入的形状与目标形状相同。为此，我们对 MNIST 数据中 10 个目标值进行了一次独热编码，这样对于每批数据，目标的形状均为 [batch_size, 10]。

如果使用 PyTorch 的 nn.CrossEntropyLoss 类（与之前的 SoftmaxCrossEntropyLoss 类完全一样），那么就不必采取这种做法。这个损失函数需要以下两个 Tensor。

- 大小为 [batch_size, num_classes] 的预测 Tensor，就像 SoftmaxCrossEntropyLoss 类之前所做的那样。
- 大小为 [batch_size] 的目标 Tensor，包含 num_classes 个值。

在前面的示例中，y_train 仅是大小为 [60000] 的数组（MNIST 训练集中的观测值的数量），而 y_test 的大小仅为 [10000]（测试集中的观测值的数量）。

就像使用 X_train、y_train、X_test 和 y_test 一样，在内存中加载整个训练集和测试集来训练模型，这种做法的内存使用率明显很低。既然要处理更大的数据集，我们应该讨论另一个最佳实践。PyTorch 有一个解决方法，即 DataLoader 类。

DataLoader类和transforms模块

回想一下，在第 4 章对 MNIST 的建模中，我们对 MNIST 中的数据应用了一个简单的预处理步骤，即减去总体均值（平均值）并除以标准偏差，从而对数据集进行大致的归一化：

```
X_train, X_test = X_train - X_train.mean(), X_test - X_train.mean()
X_train, X_test = X_train / X_train.std(), X_test / X_train.std()
```

尽管如此，这仍然需要首先将这两个数组完全读入内存。在神经网络中，随着批量数据的输入，动态执行此预处理步骤将更加高效。PyTorch 的一些内置功能可以执行此操作，这些功能通常应用于图像数据，比如通过 transforms 模块执行转换，以及通过 torch.utils.data 引入 DataLoader 类：

```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

之前，可以通过以下方式将整个训练集读入 X_train：

```
mnist_trainset = MNIST(root="../data/", train=True)
X_train = mnist_trainset.train_data
```

然后，对 X_train 执行转换，将其转换为可用于建模的形式。

PyTorch 让我们可以在读入每批数据时便捷地执行许多转换。这样既可以避免将整个数据集读取到内存中，又可以应用 PyTorch 的转换功能。

首先，定义一个转换列表，对读取的每一批数据执行转换操作。例如，以下转换将每幅 MNIST 图像都转换为一个 Tensor[4]，然后“归一化”数据集（减去总体均值，然后除以标准偏差），总体均值和标准偏差分别是 0.1305 和 0.3081。

```
img_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1305,), (0.3081,))
])
```

注 4：在默认情况下，大多数 PyTorch 数据集是“PIL 图像”。因此，transforms.ToTensor() 通常是列表中的第一个转换。



Normalize 实际上从输入图像的每个通道中减去平均值和标准偏差。因此，在处理具有 3 个输入通道的彩色图像时，通常会对两个元组（每个具有 3 个数字）执行 Normalize 转换。例如，执行 transforms.Normalize((0.1, 0.3, 0.6), (0.4, 0.2, 0.5))，它将告诉 DataLoader 类以下信息。

- 使用平均值 0.1 和标准偏差 0.4 归一化第 1 个通道。
- 使用平均值 0.3 和标准偏差 0.2 归一化第 2 个通道。
- 使用平均值 0.6 和标准偏差 0.5 归一化第 3 个通道。

其次，一旦应用了这些转换，就在批量读取数据时将它们应用于 dataset：

```
dataset = MNIST("../mnist_data/", transform=img_transforms)
```

接着，定义 DataLoader 类来接收此数据集，并定义用于连续生成批量数据的规则：

```
dataloader = DataLoader(dataset, batch_size=60, shuffle=True)
```

最后，修改 Trainer 类，从而使用 dataloader 生成用于训练神经网络的批量数据，而不是像以前一样将整个数据集加载到内存中并使用 batch_generator 函数手动生成它们。在本书的随书文件中，我展示了一个使用 DataLoader 类训练 CNN 的示例。Trainer 中的主要变化是改变了下面这一行代码：

```
for X_batch, y_batch in enumerate(batch_generator):
```

下面是修改后的代码：

```
for X_batch, y_batch in enumerate(train_dataloader):
```

此外，现在不再将整个训练集输入 fit 函数，而是输入到 DataLoader 类中：

```
trainer.fit(train_dataloader = train_loader,
            test_dataloader = test_loader,
            epochs=1,
            eval_every=1)
```

正如刚才所做的那样，使用这种架构并调用 fit 可以使 MNIST 在一轮训练后达到约 97% 的准确度。但是，比准确性更重要的是，你已经了解了如何在高性能框架中实现我们从基本原理中得出的概念。了解了底层概念和框架之后，你可以去修改本书中的代码，并尝试其他卷积架构和数据集。

如前所述，CNN 是一种高级架构。下面转向另一种高级架构 LSTM，即本书中最高级的 RNN 变体，并展示如何在 PyTorch 中实现它。

7.4 PyTorch中的LSTM

第 6 章介绍了如何从零开始编写 LSTM。我们对 LSTMLayer 类进行了编码，从而接收一个大小为 [batch_size， sequence_length， feature_size] 的输入 ndarray，并输出一个大小为 [batch_size， sequence_length， feature_size] 的 ndarray。此外，每层都接收一个隐藏状态和一个单元状态，每个状态的形状都初始化为 [1，hidden_size]。在传入一批数据时，形状会扩展为 [batch_size，hidden_size]；在迭代完成之后，形状变回 [1，hidden_size]。

因此，LSTMLayer 类的 __init__ 方法定义如下：

```
class LSTMLayer(PyTorchLayer):
    def __init__(self,
                 sequence_length: int,
                 input_size: int,
                 hidden_size: int,
                 output_size: int) -> None:
        super().__init__()
        self.hidden_size = hidden_size
        self.h_init = torch.zeros((1, hidden_size))
        self.c_init = torch.zeros((1, hidden_size))
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = DenseLayer(hidden_size, output_size)
```

与卷积层一样，PyTorch 中有一个名为 nn.lstm 的运算，可用于实现 LSTM。注意，在自定义 LSTMLayer 类中，self.fc 属性存储了 DenseLayer 类。你可能还记得，在第 6 章中，LSTM 单元的最后一步是将最终隐藏状态输入到 Dense 层的运算中（权重乘法和添加偏差），从而为每个运算将隐藏状态的维度转换为 output_size。PyTorch 的处理方式有所不同：nn.lstm 运算仅在每个时间步长中输出隐藏状态。因此，和神经网络层一样，为了让 LSTMLayer 类输出的维度与其输入的维度不同，我们在最后添加一个 Dense 层，从而将隐藏状态的维度转换为 output_size。

通过这种修改，forward 方法现在变得简单明了，类似于第 6 章的 LSTMLayer 类中的 forward 方法：

```
def forward(self, x: Tensor) -> Tensor:

    batch_size = x.shape[0]

    h_layer = self._transform_hidden_batch(self.h_init,
                                           batch_size,
                                           before_layer=True)
    c_layer = self._transform_hidden_batch(self.c_init,
                                           batch_size,
                                           before_layer=True)

    x, (h_out, c_out) = self.lstm(x, (h_layer, c_layer))
```

```
        self.h_init, self.c_init = (
            self._transform_hidden_batch(h_out,
                                         batch_size,
                                         before_layer=False).detach(),
            self._transform_hidden_batch(c_out,
                                         batch_size,
                                         before_layer=False).detach()
                                        )

        x = self.fc(x)

        return x
```

考虑到第 6 章中的 LSTM 实现，这里的关键代码行看起来应该很熟悉：

```
x, (h_out, c_out) = self.lstm(x, (h_layer, c_layer))
```

除此之外，还利用辅助函数 `self._transform_hidden_batch` 对 `self.lstm` 前后的隐藏状态以及单元状态进行了重塑。可以在本书的随书文件中查看完整代码。

最后，将模型包装起来很容易，如下所示。

```
class NextCharacterModel(PyTorchModel):
    def __init__(self,
                 vocab_size: int,
                 hidden_size: int = 256,
                 sequence_length: int = 25):
        super().__init__()
        self.vocab_size = vocab_size
        self.sequence_length = sequence_length

        # 这个模型只有一层，该层的输出大小与输入大小相同
        self.lstm = LSTMLayer(self.sequence_length,
                              self.vocab_size,
                              hidden_size,
                              self.vocab_size)
    def forward(self,
                inputs: Tensor):
        assert_dim(inputs, 3) # batch_size, sequence_length, vocab_size

        out = self.lstm(inputs)

        return out.permute(0, 2, 1)
```



`nn.CrossEntropyLoss` 函数期望前两个维度是 `batch_size` 和类的分布。然而，按照实现 LSTM 的方式，我们将类的分布作为 `LSTMLayer` 类中的最后一个维度（`vocab_size`）。因此，为了将最终的模型输出传递给损失，这里使用 `out.permute(0,2,1)` 将包含字母分布的维度移到了第 2 个维度。

本书的随书文件展示了如何编写从 `PyTorchTrainer` 类继承的 `LSTMTrainer` 类，并使用它来训练 `NextCharacterModel` 生成文本。与第 6 章一样，这里同样进行文本预处理：选择文本

序列，对字母进行独热编码，然后对独热编码的字母序列进行分组，从而形成批次。

以上总结了如何在 PyTorch 中实现用于监督学习的神经网络架构，包括全连接神经网络、CNN 和 RNN。最后，本书将简要介绍如何将神经网络用于另一种机器学习：**无监督学习**（unsupervised learning）。

7.5 后记：通过自编码器进行无监督学习

本书始终专注于如何使用深度学习模型来解决涉及**监督学习**的问题。当然，机器学习还包括无监督学习，通常描述为“在没有标签的数据中寻找结构”。但是，我喜欢将无监督学习描述为发现尚未测量的数据特征之间的关系，而监督学习则需要发现已测量数据特征之间的关系。

假设你有一个没有标签的图像数据集。你对这些图像不太了解，比如说，你不知道其中有几个数字，并且你想知道以下问题的答案。

- 有多少个数字？
- 哪些数字看起来彼此相似？
- 是否存在与其他图像明显不同的“异常”图像？

要理解如何利用深度学习来解决这些问题，必须从概念角度思考深度学习模型的工作机制。

7.5.1 表征学习

深度学习模型可以通过学习做出准确的预测。模型通过将接收到的输入转换成表征来实现这一点，这种表征更为抽象，也更易于调整，从而可以直接对相关问题进行预测。特别是在神经网络的最后一层，即紧靠预测本身的那一层[5]，神经网络试图基于输入数据创建对预测任务尽可能有用的表征，如图 7-1 所示。

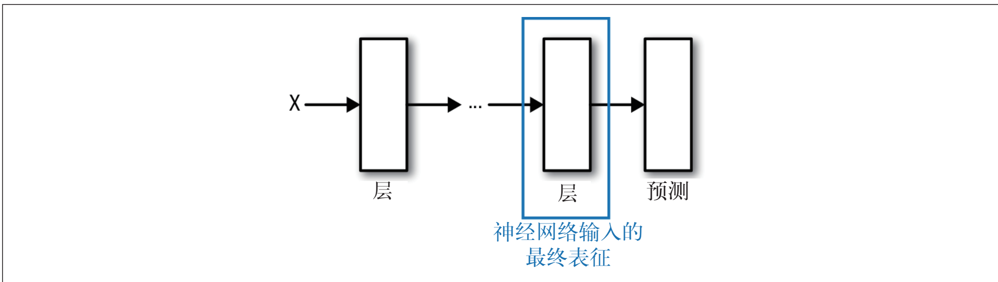


图 7-1：神经网络的最后一层（预测之前的那一层），代表了神经网络对输入的表征，对预测任务最有帮助

注 5：对于回归问题，只有一个神经元，而对于分类问题，有 `num_classes` 个神经元。

一旦训练完成，模型不仅可以对新的数据点进行预测，还可以生成这些数据点的表征。除了预测，还可以将它们用于聚类、相似性分析或异常点检测。

7.5.2　应对无标签场景的方法

整个方法的局限性在于，需要标签来训练模型，从而生成表征。问题是，如何在没有任何标签的情况下训练模型，从而生成“有用”的表征？如果没有标签，则需要使用唯一拥有的东西，即训练数据本身。这是一种叫作**自编码器**（autoencoder）的神经网络架构的理念，该架构涉及训练神经网络来**重建**（reconstruct）训练数据，从而迫使神经网络学习对重建最有帮助的每个数据点的表征。

示意图

图 7-2 描绘了自编码器的总体情况。

1. 一组层将数据转换为数据的压缩表征。
2. 另一组层将此表征转换为大小和形状与原始数据相同的输出。

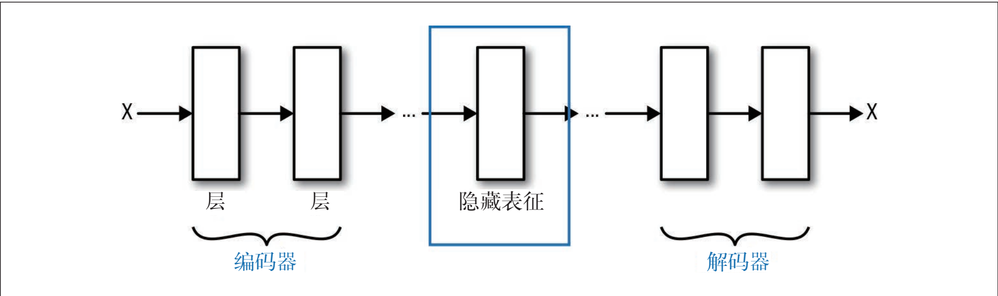


图 7-2：自编码器具有一组将输入映射到低维表征的层（可以称为“编码器”），以及另一组将低维表征映射回输入的层（可以称为“解码器”）。这种架构迫使神经网络学习对重建输入最有用的低维表征

实现这种架构需要用到 PyTorch 的一些特性，目前本书暂未涉及这些特性。

7.5.3　在PyTorch中实现自编码器

现在展示一个简单的自编码器，该编码器接收一幅输入图像，将其输入到两个卷积层和一个 `Dense` 层中来生成表征，然后再将这个表征传递回 `Dense` 层和卷积层来生成一个与输入大小相同的输出。我们将以此为例说明在 PyTorch 中实现高级架构时所采用的两种常见做法。首先，可以在一个 `PyTorchModel` 类中包含多个 `PyTorchModel` 类作为自身的属性，就像之前将 `PyTorchLayer` 类定义为此类模型的属性一样。以下示例将实现自编码器，其中有两个 `PyTorchModel` 类作为属性，分别是 `Encoder` 类和 `Decoder` 类。一旦训练了模型，就可以将经过训练的 `Encoder` 类作为自己的模型来生成表征。

下面是 Encoder 类的定义：

```
class Encoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Encoder, self).__init__()
        self.conv1 = ConvLayer(1, 14, activation=nn.Tanh())
        self.conv2 = ConvLayer(14, 7, activation=nn.Tanh(), flatten=True)

        self.dense1 = DenseLayer(7 * 28 * 28, hidden_dim, activation=nn.Tanh())

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        x = self.conv1(x)
        x = self.conv2(x)
        x = self.dense1(x)

        return x
```

下面是 Decoder 类的定义。

```
class Decoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Decoder, self).__init__()
        self.dense1 = DenseLayer(hidden_dim, 7 * 28 * 28, activation=nn.Tanh())

        self.conv1 = ConvLayer(7, 14, activation=nn.Tanh())
        self.conv2 = ConvLayer(14, 1, activation=nn.Tanh())

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 2)

        x = self.dense1(x)

        x = x.view(-1, 7, 28, 28)
        x = self.conv1(x)
        x = self.conv2(x)

        return x
```



如果所使用的步幅大于 1，就不能像在这里一样简单地使用常规卷积将编码转换为输出，而是必须使用**转置卷积**（transposed convolution），其中该运算输出的图像大小将大于输入的图像大小。要了解更多相关信息，参见 PyTorch 文档中的 `nn.ConvTranspose2d` 运算。

然后，Autoencoder 类本身可以将这些包装起来，其定义如下：

```
class Autoencoder(PyTorchModel):
    def __init__(self,
```

```
                 hidden_dim: int = 28):
        super(Autoencoder, self).__init__()

        self.encoder = Encoder(hidden_dim)

        self.decoder = Decoder(hidden_dim)

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        encoding = self.encoder(x)
        x = self.decoder(encoding)

        return x, encoding
```

Autoencoder 类中的 forward 方法展示了 PyTorch 的第二种常见做法：由于最终希望看到模型生成的隐藏表征，因此 forward 方法返回两个元素，即 encoding 以及用于训练神经网络的输出 x。

当然，必须修改 Trainer 类来适应这种情况。具体地说，当前编写的 PyTorchModel 类从其 forward 方法仅输出一个 Tensor。事实证明，当修改 Trainer 类之后，它默认会返回一个 Tensor 元组（即使该元组的长度为 1）。这非常有用，我们能够据此轻松编写像 Autoencoder 类这样的模型，并且整个过程易于实现。只需完成 3 件易事即可。首先，创建 PyTorchModel 基类中 forward 方法的签名：

```
def forward(self, x: Tensor) -> Tuple[Tensor]:
```

并且在从 PyTorchModel 基类继承的所有模型的 forward 方法末尾，写下 return x, encoding，而不是像以前那样执行 return x。

然后，修改 Trainer 类，使其始终将模型返回的第一个元素作为输出：

```
output = self.model(X_batch)[0]
...
output = self.model(X_test)[0]
```

Autoencoder 模型的另一个显著特征，就是在最后一层应用了 Tanh 激活函数，这意味着模型的输出将介于 –1 和 1 之间。对于任何模型，其输出都应该和所比较的目标大小相同，这里，目标就是输入本身。因此，应该将输入的范围缩放到 –1 ～ 1，如以下代码所示：

```
X_train_auto = (X_train - X_train.min())
                / (X_train.max() - X_train.min()) * 2 - 1
X_test_auto = (X_test - X_train.min())
                / (X_train.max() - X_train.min()) * 2 - 1
```

最后，可以使用训练代码来训练模型，该代码现在看起来应该很熟悉（这里随机使用 28 作为编码输出的维数）：

```
model = Autoencoder(hidden_dim=28)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

trainer = PyTorchTrainer(model, optimizer, criterion)

trainer.fit(X_train_auto, X_train_auto,
            X_test_auto, X_test_auto,
            epochs=1,
            batch_size=60)
```

一旦运行此代码并训练了模型，就可以将 X_test_auto 输入到模型中，从而查看重建的图像和图像表征，这是因为定义的 forward 方法会返回两个量：

```
reconstructed_images, image_representations = model(X_test_auto)
```

每个 reconstructed_images 元素都是一个形状为 [1, 28, 28] 的 Tensor，代表神经网络在将原始图像输入到自编码器架构后，也就是强制将图像穿过较低维度的层，对其进行重建的最佳尝试效果。图 7-3 显示了随机选择的重建图像以及原始图像。

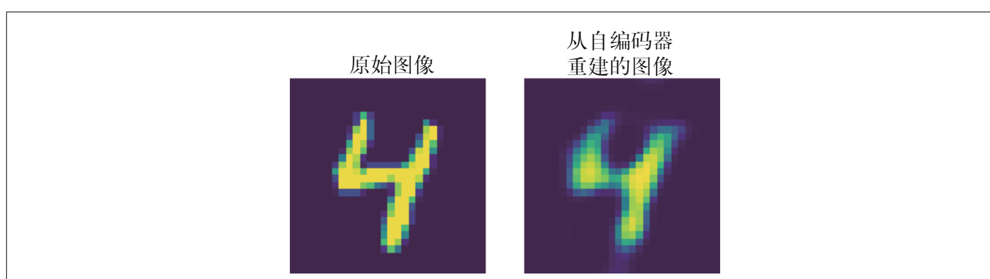


图 7-3：来自 MNIST 测试集的图像，以及输入到自编码器后的重建图像

从视觉上看，这两幅图像看起来很相似，这告诉我们神经网络似乎获取了原始图像（784 像素），并将其映射到较低维度的空间（28 维），造成关于原始图像的大部分信息编码在这个长度为 28 的向量中。在没有看到标签的情况下，如何通过检查整个数据集，来检查神经网络是否学会了图像数据结构呢？这里的“数据结构”意味着底层数据实际上是 10 幅手写数字的图像。在理想情况下，新的 28 维空间中接近给定图像的图像应该为同一数字，或者至少在视觉上非常相似，这是因为视觉相似性是人类区分不同图像的方式。可以通过降维技术来检验这种情况，即 t-SNE[6]。t-SNE 的降维方式类似于训练神经网络的方式：它从初始的低维表征开始，然后对其进行更新，这样随着时间的推移，它利用高维空间中“相互靠近”的点会在低维空间中“相互靠近”这一性质（反之亦然）来逼近一个解[7]。

注 6：全称为 t-Distributed Stochastic Neighbor Embedding，t 分布随机邻域嵌入。Laurens van der Maaten 在就读研究生期间发明了该技术，当时他的导师是 Geoffrey Hinton（神经网络的“奠基人”之一）。

注 7：参见 Laurens van der Maaten 和 Geoffrey Hinton 在 2008 年共同发表的论文“Visualizing Data using t-SNE”。

我们将尝试下面两种方法。

1. 将 10 000 幅图像输入 t-SNE，并将维度降为 2。
2. 可视化所得的二维空间，并根据数据点的实际标签（自编码码器看不到）为它们着色。

结果如图 7-4 所示。

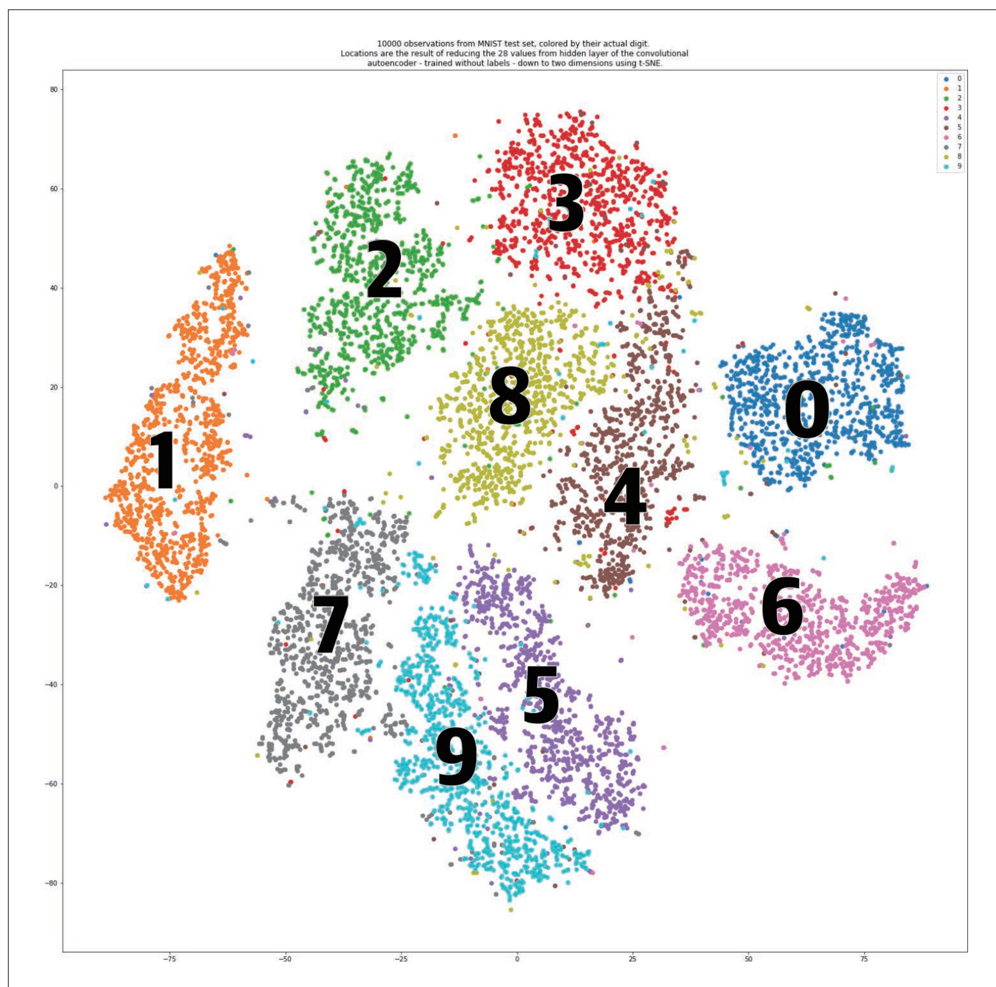


图 7-4：针对自编码器的 28 维学习空间运行 t-SNE 的结果[8]

似乎每个数字的图像在很大程度上都归为一个独立的类别。这一点表明，在没有看到任何标签的情况下，通过训练自编码器架构学习仅从低维表征来重构原始图像，确实能让它发

注 8：可以登录图灵社区查看彩图：ituring.cn/book/2759。——编者注

现这些图像的底层结构[9]。10 个数字不仅被表示为不同的类别，在视觉上相似的数字也更靠近：在顶部和靠右的位置，有数字 3、4 和 8；在底部，可以看到 5 和 9 紧密地聚在一起，7 也离得不远。差别最大的数字（0、1 和 6）与其余数字相距最远。

7.5.4 更强大的无监督学习测试及解决方案

7.5.3 节展示了一个很简单的例子，用于测试模型是否已经学习了输入图像空间的底层结构。至此，基于视觉上相似的图像具有相似表征的这一特性，CNN 可以学习数字图像的表征也就不足为奇了。一个更强大的测试将能够检查神经网络是否发现了“平滑”的底层空间：在该空间中，除了通过编码器网络输入的真实数字生成的向量，所有长度为 28 的向量也都可以映射到逼真的数字。事实证明，自编码器无法实现这一点。下面生成 5 个长度为 28 的随机向量并将其输入到解码器网络中，图 7-5 显示了执行结果，其中 `Autoencoder` 类的属性包含一个 `Decoder` 类：

```
test_encodings = np.random.uniform(low=-1.0, high=1.0, size=(5, 28))
test_imgs = model.decoder(Tensor(test_encodings))
```

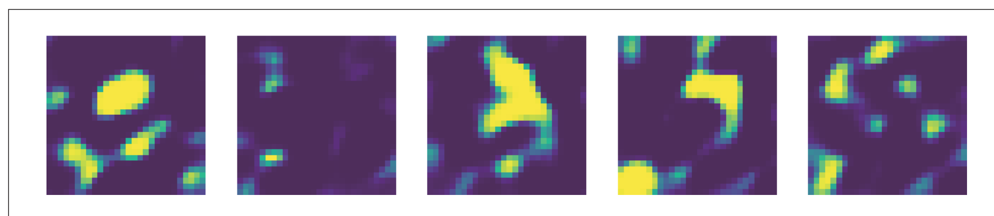


图 7-5：将 5 个随机向量输入解码器的结果

可以看到，这里生成的图像看起来不像数字。因此，尽管自编码器可以用合理的方式将数据映射到低维空间，但似乎无法学习前面提到的“平滑”空间。

解决该问题的方法是，通过训练神经网络，让其学会在一个“平滑”的底层空间中表示训练集中的图像，这是 GAN[10] 的主要成就之一。GAN 诞生于 2014 年，通过一个能够同时训练两个神经网络的训练程序，它实现了让神经网络生成逼真的图像，也正是因为这一点而广为人知。在 2015 年，GAN 确实取得了长足的进步，当研究人员在两个神经网络中同时使用 GAN 与深层卷积架构时，不仅可以生成看起来逼真的卧室彩色图像（64 像素 ×64 像素），还可以从随机生成的 100 维向量中生成这些图像的大量样本[11]。这表明神经网络确

注 9：此外，我们无须过多尝试即可做到这一点：这里的架构非常简单，并且由于只训练一轮，因此没有使用前面讨论的训练神经网络的技术（例如学习率衰减）。这说明，使用类似自编码器的架构来学习不带标签的数据集结构，这种想法很好，而不仅仅是“碰巧奏效”。

注 10：generative adversarial network，生成对抗网络。

注 11：参见 Alec Radford 等人所发表的有关 DCGAN 的论文“Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”，以及 PyTorch 的相关文档。

实已经了解了这些未标记图像的底层表征。GAN 涉及的内容足以另写一本书，这里只对它进行大致介绍，并不包括具体实现细节。

7.6 小结

至此，你已经深入了解了一些较为流行的高级深度学习架构的实现机制，以及如何基于较为流行的高性能深度学习框架来实现这些架构。现在就只差实践了，只有通过实践，才能使用深度学习模型解决现实世界中的问题。幸运的是，你可以采用最简单的做法，那就是阅读其他人的代码，快速掌握细节和实现技巧，从而构建能够处理某些问题的模型架构。本书的随书文件列出了一些后续步骤，在实践过程中可以参考。加油！

附录

深入探讨

为了完善前面的内容，本附录将深入探讨一些技术问题，这些问题较为重要，可以在实践中参考。

矩阵链式法则

首先来看一下第 1 章中的链式法则表达式，其中用 $\boldsymbol{W}^{\mathrm{T}}$ 代替 $\frac{\partial v}{\partial u}(\boldsymbol{X})$ 有以下原因。

注意，L 的计算公式如下：

$$\sigma(XW_{11})+\sigma(XW_{12})+\sigma(XW_{21})+\sigma(XW_{22})+\sigma(XW_{31})+\sigma(XW_{32})$$

其中，$\sigma(XW_{11})$ 和 $\sigma(XW_{12})$ 的计算方法展开为：

$$\sigma(XW_{11})=\sigma(x_{11}\times w_{11}+x_{12}\times w_{21}+x_{13}\times w_{31})$$
$$\sigma(XW_{12})=\sigma(x_{11}\times w_{12}+x_{12}\times w_{22}+x_{13}\times w_{32})$$

以此类推。现在聚焦其中一个表达式，如果对 $\boldsymbol{X}$ 的每一个元素（L 计算公式中的 6 个组成部分），比如 $\sigma(XW_{11})$，求其偏导数，会发生什么情况？

从 $\sigma(XW_{11})$ 的计算公式可以看到，通过简单应用链式法则，很容易计算 x_1 的偏导数：

$$\frac{\partial\sigma}{\partial u}(XW_{11})\times w_{11}$$

在 XW_{11} 的表达式中，由于只有 w_{11} 与 x_{11} 相乘，因此相对于其他所有值，偏导数都是 0。

这样一来，计算 $\sigma(XW_{11})$ 相对于 $\boldsymbol{X}$ 的所有元素的偏导数，可以得出 $\frac{\partial\sigma(XW_{11})}{\partial\boldsymbol{X}}$ 整体的表达式，如下所示：

$$\frac{\partial\sigma(XW_{11})}{\partial\boldsymbol{X}}=\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{11})\times w_{11} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{21} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{31}\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}$$

同理，可以计算 $\sigma(XW_{32})$ 相对于 $\boldsymbol{X}$ 的每个元素的偏导数：

$$\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{32}\end{bmatrix}$$

现在我们拥有了直接计算 $\frac{\partial\Lambda}{\partial\boldsymbol{X}}(S)$ 的所有组件。可以简单地计算 6 个与前面的矩阵形式相同的矩阵，并将结果相加。

注意，这里虽然没有涉及高等数学运算，但过程还是容易变乱。这个表达式很简单，你可以跳过下面的计算过程，直接查看结论。但是，理解整个计算过程之后，你会对结论产生更全面的认识，那就是它真的非常简单。不过，这不就是实践的意义吗？

这里只有两个步骤。首先，明确写出 $\frac{\partial\Lambda}{\partial\boldsymbol{X}}(S)$ 是上述 6 个矩阵的总和：

$$\frac{\partial\Lambda}{\partial\boldsymbol{X}}(S)=\frac{\partial\sigma(XW_{11})}{\partial\boldsymbol{X}}+\frac{\partial\sigma(XW_{12})}{\partial\boldsymbol{X}}+\frac{\partial\sigma(XW_{21})}{\partial\boldsymbol{X}}+\frac{\partial\sigma(XW_{22})}{\partial\boldsymbol{X}}+\frac{\partial\sigma(XW_{31})}{\partial\boldsymbol{X}}+\frac{\partial\sigma(XW_{32})}{\partial\boldsymbol{X}}=$$

$$\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{11})\times w_{11} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{21} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{31}\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}+$$

$$\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{12})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{12})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{12})\times w_{32}\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}+$$

$$\begin{bmatrix} 0 & 0 & 0 \\ \frac{\partial\sigma}{\partial u}(XW_{21})\times w_{11} & \frac{\partial\sigma}{\partial u}(XW_{21})\times w_{21} & \frac{\partial\sigma}{\partial u}(XW_{21})\times w_{31} \\ 0 & 0 & 0 \end{bmatrix}+$$

$$\begin{bmatrix} 0 & 0 & 0 \\ \frac{\partial\sigma}{\partial u}(XW_{22})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{22})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{22})\times w_{32} \\ 0 & 0 & 0 \end{bmatrix}+$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{\partial\sigma}{\partial u}(XW_{31})\times w_{11} & \frac{\partial\sigma}{\partial u}(XW_{31})\times w_{21} & \frac{\partial\sigma}{\partial u}(XW_{31})\times w_{31} \end{bmatrix}+$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{32} \end{bmatrix}$$

现在，我们将该总和合并为一个大矩阵。该矩阵不会立即具有任何直观的形式，但实际上是对前面总和的计算结果：

$$\frac{\partial\Lambda}{\partial \boldsymbol{X}}(S)=$$

$$\begin{bmatrix} \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{11}+\frac{\partial\sigma}{\partial u}(XW_{12})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{21}+\frac{\partial\sigma}{\partial u}(XW_{12})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{31}+\frac{\partial\sigma}{\partial u}(XW_{12})\times w_{32} \\ \frac{\partial\sigma}{\partial u}(XW_{21})\times w_{11}+\frac{\partial\sigma}{\partial u}(XW_{22})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{21})\times w_{21}+\frac{\partial\sigma}{\partial u}(XW_{22})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{21})\times w_{31}+\frac{\partial\sigma}{\partial u}(XW_{22})\times w_{32} \\ \frac{\partial\sigma}{\partial u}(XW_{31})\times w_{11}+\frac{\partial\sigma}{\partial u}(XW_{32})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{31})\times w_{21}+\frac{\partial\sigma}{\partial u}(XW_{32})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{31})\times w_{31}+\frac{\partial\sigma}{\partial u}(XW_{32})\times w_{32} \end{bmatrix}$$

然后就是精彩的部分。回想一下，第 1 章提到了下面这个公式：

$$\boldsymbol{W}=\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

可以看到，$\boldsymbol{W}$ 隐藏在前面的矩阵中，其实它只是被转置了。再回想一下这个公式：

$$\frac{\partial \Lambda}{\partial u}(S) = \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix}$$

事实证明，原来的矩阵可以用下面这种形式表示：

$$\frac{\partial \Lambda}{\partial \boldsymbol{X}}(\boldsymbol{X}) = \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} = \frac{\partial \Lambda}{\partial u}(S) \times \boldsymbol{W}^{\mathrm{T}}$$

注意，我们正在试图替换下面这个等式中的问号：

$$\frac{\partial \Lambda}{\partial \boldsymbol{X}}(\boldsymbol{X}) = \frac{\partial \Lambda}{\partial u}(S) \times ?$$

很明显，问号就是 $\boldsymbol{W}$。

还要注意，这与之前看到的一维结果相同。同样，这既解释了深度学习起作用的原因，又让我们能够清晰地实现深度学习模型。这是否意味着我们可以替换前面等式中的问号，将该等式理解成 $\frac{\partial v}{\partial \boldsymbol{X}}(\boldsymbol{X},\boldsymbol{W}) = \boldsymbol{W}^{\mathrm{T}}$？不，不完全是。但是，如果将两个输入（$\boldsymbol{X}$ 和 $\boldsymbol{W}$）相乘获得结果 N，并将这些输入传递给某个非线性函数 σ，从而获取输出 S，那么可以这样表达：

$$\frac{\partial \sigma}{\partial \boldsymbol{X}}(\boldsymbol{X},\boldsymbol{W}) = \frac{\partial \sigma}{\partial u}(N) \times \boldsymbol{W}^{\mathrm{T}}$$

这样一来，我们便可以使用矩阵乘法有效地计算并表达梯度更新。另外，通过类似的计算过程，可以得出下面这个公式。

$$\frac{\partial \sigma}{\partial \boldsymbol{W}}(\boldsymbol{X},\boldsymbol{W}) = \boldsymbol{X}^{\mathrm{T}} \times \frac{\partial \sigma}{\partial u}(N)$$

关于偏差项的损失梯度

在全连接神经网络中计算相对于偏差项的损失导数时，为什么要沿 $axis = 0$ 进行求和？接下来将对这个问题展开讨论。

当发生下面这种情况时，要在神经网络中添加一个偏差项：假设有一批数据，这些数据由维度为 n 行（批次大小）乘以 f 列（特征数量）的矩阵表示，我们向每个 f 特征都添加一个数字。例如，第 2 章的神经网络示例有 13 个特征，偏差项 B 有 13 个数字。第 1 个数字将添加到 `M1 = np.dot(X, weights[W1])` 的第 1 列的每一行，第 2 个数字将添加到第 2 列的每一行，以此类推。另外，在神经网络中，`B2` 将包含一个数字，该数字会直接添加到 `M2` 每一列的每一行中。这样一来，由于矩阵的每一行都添加了相同的数字，因此在后向传递时，需要沿某个维度对梯度进行求和。这就是为什么我们沿 `axis=0` 对 `dLdB1` 和 `dLdB2` 的表达式进行求和，例如，`dLdB1 = (dLdN1 × dN1dB1).sum(axis=0)`。图 A-1 是对这些内容的可视化说明，其中附有一些注释。

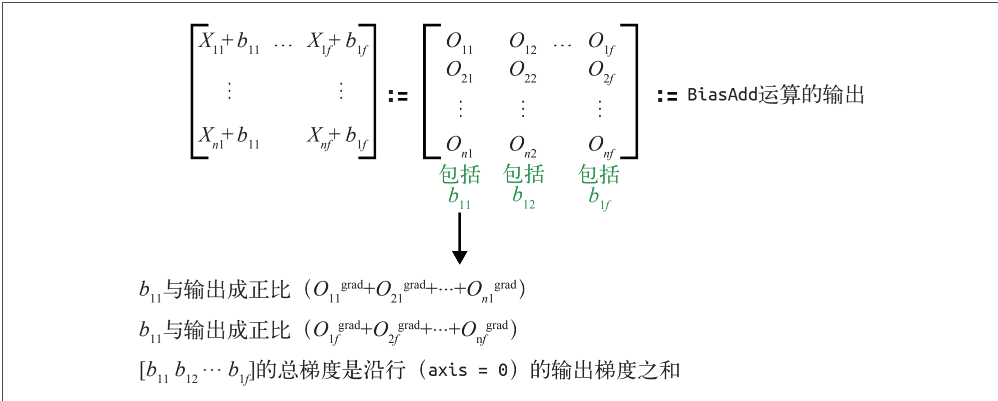


图 A-1：在计算全连接层的输出相对于偏差的导数时，关于沿 axis = 0 求和部分的说明

通过矩阵乘法进行卷积

本节将展示如何用批量矩阵乘法来表示批次和多通道卷积运算，从而在 NumPy 库中高效地实现它。

要了解卷积的工作机制，可以参考全连接神经网络在前向传递中执行的操作。

1. 接收大小为 `[batch_size, in_features]` 的输入。
2. 将其乘以大小为 `[in_features, out_features]` 的参数。
3. 得到大小为 `[batch_size, out_features]` 的输出。

卷积层执行的操作则不同，如下所述。

1. 接收大小为 `[batch_size, in_channels, img_height, img_width]` 的输入。
2. 使用大小为 `[in_channels, out_channels, param_height, param_width]` 的参数对其进行卷积。
3. 得到大小为 `[batch_size, in_channels, img_height, img_width]` 的输出。

使卷积运算看起来更像常规前馈运算的关键是首先**从输入图像的每个通道中提取** img_height×img_width **个“图像块”**。一旦提取了这些图像块，就可以重塑输入，从而使用 NumPy 库中的 `np.matmul` 函数将卷积运算表示为一个批量矩阵乘法。具体操作如下。

```
def _get_image_patches(imgs_batch: ndarray,
                       fil_size: int):
    '''
    imgs_batch: [batch_size, channels, img_width, img_height]
    fil_size: int
    '''
    # 填充图像
    imgs_batch_pad = np.stack([_pad_2d_channel(obs, fil_size // 2)
                              for obs in imgs_batch])
    patches = []
    img_height = imgs_batch_pad.shape[2]

    # 针对图像中的每个位置执行的操作
    for h in range(img_height-fil_size+1):
        for w in range(img_height-fil_size+1):

            # 获取大小为[fil_size, fil_size]的图像块
            patch = imgs_batch_pad[:, :, h:h+fil_size, w:w+fil_size]
            patches.append(patch)

    # 堆叠，获取一个大小为
    # [img_height * img_width, batch_size, n_channels, fil_size, fil_size]的输出
    return np.stack(patches)
```

然后，可以通过以下方式计算卷积运算的输出。

1. 获取大小为 `[batch_size, in_channels, img_height x img_width, filter_size, filter_size]` 的图像块。
2. 将其重塑为 `[batch_size, img_height × img_width, in_channels × filter_size × filter_size]`。
3. 将参数重塑为 `[in_channels × filter_size × filter_size, out_channels]`。
4. 进行批量矩阵乘法后，结果将为 `[batch_size, img_height × img_width, out_channels]`。
5. 将该结果重塑为 `[batch_size, out_channels, img_height, img_width]`。

```
def _output_matmul(input_: ndarray,
                   param: ndarray) -> ndarray:
    '''
    conv_in: [batch_size, in_channels, img_width, img_height]
    param: [in_channels, out_channels, fil_width, fil_height]
    '''

    param_size = param.shape[2]
    batch_size = input_.shape[0]
    img_height = input_.shape[2]
    patch_size = param.shape[0] * param.shape[2] * param.shape[3]
```

```
    patches = _get_image_patches(input_, param_size)

    patches_reshaped = (
        patches
        .transpose(1, 0, 2, 3, 4)
        .reshape(batch_size, img_height * img_height, -1)
        )

    param_reshaped = param.transpose(0, 2, 3, 1).reshape(patch_size, -1)

    output = np.matmul(patches_reshaped, param_reshaped)

    output_reshaped = (
        output
       .reshape(batch_size, img_height, img_height, -1)
       .transpose(0, 3, 1, 2)
    )

    return output_reshaped
```

那是前向传递！对于后向传递，必须同时计算参数梯度和输入梯度。同样，我们可以借鉴全连接神经网络中完成此运算的方式。从参数梯度开始，全连接神经网络的梯度为：

```
np.matmul(self.inputs.transpose(1, 0), output_grad)
```

可以就此推算卷积操作中后向传递的实现方式。这里，输入形状为 [batch_size, in_channels, img_height, img_width]，接收到的输出梯度为 [batch_size, out_channels, img_height, img_width]。考虑到参数的形状为 [in_channels, out_channels, param_height, param_width]，可以通过以下步骤实现这个转换。

1. 必须从输入图像中提取图像块，从而获得与上次相同的输出，其形状为 [batch_size, in_channels, img_height x img_width, filter_size, filter_size]。
2. 与全连接场景中的乘法类似，将其形状重塑为 [in_channels × param_height × param_width, batch_size × img_height × img_width]。
3. 重塑输出形状，即由最初的 [batch_size, out_channels, img_height, img_width] 变为 [batch_size × img_height × img_width, out_channels]。
4. 将它们相乘，得到形状为 [in_channels × param_height × param_width, out_channels] 的输出。
5. 重塑这个输出以获得形状为 [in_channels, out_channels, param_height, param_width] 的最终参数梯度。

具体实现过程如下：

```
def _param_grad_matmul(input_: ndarray,
                       param: ndarray,
                       output_grad: ndarray):
```

```
    '''
    input_: [batch_size, in_channels, img_width, img_height]
    param: [in_channels, out_channels, fil_width, fil_height]
    output_grad: [batch_size, out_channels, img_width, img_height]
    '''

    param_size = param.shape[2]
    batch_size = input_.shape[0]
    img_size = input_.shape[2] ** 2
    in_channels = input_.shape[1]
    out_channels = output_grad.shape[1]
    patch_size = param.shape[0] * param.shape[2] * param.shape[3]
    patches = _get_image_patches(input_, param_sizes)

    patches_reshaped = (
        patches
        .reshape(batch_size * img_size, -1)
        )

    output_grad_reshaped = (
        output_grad
        .transpose(0, 2, 3, 1)
        .reshape(batch_size * img_size, -1)
    )

    param_reshaped = param.transpose(0, 2, 3, 1).reshape(patch_size, -1)

    param_grad = np.matmul(patches_reshaped.transpose(1, 0),
                           output_grad_reshaped)

    param_grad_reshaped = (
        param_grad
        .reshape(in_channels, param_size, param_size, out_channels)
        .transpose(0, 3, 1, 2)
    )

    return param_grad_reshaped
```

另外，这里模拟全连接层中的运算，通过遵循一组相似的步骤来获得输入梯度：

```
np.matmul(output_grad, self.param.transpose(1, 0))
```

以下代码计算输入梯度：

```
def _input_grad_matmul(input_: ndarray,
                       param: ndarray,
                       output_grad: ndarray):

    param_size = param.shape[2]
    batch_size = input_.shape[0]
    img_height = input_.shape[2]
    in_channels = input_.shape[1]

    output_grad_patches = _get_image_patches(output_grad, param_size)
```

```
        output_grad_patches_reshaped = (
            output_grad_patches
            .transpose(1, 0, 2, 3, 4)
            .reshape(batch_size * img_height * img_height, -1)
        )

        param_reshaped = (
            param
            .reshape(in_channels, -1)
        )

        input_grad = np.matmul(output_grad_patches_reshaped,
                               param_reshaped.transpose(1, 0))

        input_grad_reshaped = (
            input_grad
            .reshape(batch_size, img_height, img_height, 3)
            .transpose(0, 3, 1, 2)
        )

        return input_grad_reshaped
```

3 个方法构成了 `Conv2D` 运算的核心，它们分别是 `_output`、`_param_grad` 和 `_input_grad`，可以在随书文件提供的 lincoln 库中找到它们。

关于作者

塞思·韦德曼（Seth Weidman）是一位数据科学家，具有多年应用和教授机器学习概念的经验。他最初在 Trunk Club 担任数据科学家，在那里建立了业界领先的评分模型和推荐系统。之后，他加入 Facebook 的数据科学团队，目前就职于 SentiLink，为其基础设施团队构建机器学习模型。在此期间，他为训练营和 Metis 公司培训团队教授数据科学和机器学习。他热衷于通过简单明了的方式解释复杂的概念，力求透过复杂表象，发掘简单本质。

关于封面

本书封面上的动物是北非石鸡（Alectoris barbara）。它是一种鸟，分布在非洲北部及欧洲部分地区，如加那利群岛和直布罗陀等地，生活在森林、灌木丛和干旱环境中。现在，也可以在葡萄牙、意大利和西班牙找到其踪迹。

北非石鸡体形圆胖，重达约 0.45 千克，翼展约 45.72 厘米。相比飞行，它们更喜欢步行。它们的脖颈呈红褐色，带有白色斑点，腹部呈浅黄色，侧翼有白棕相间的条纹状羽毛，腿、喙和眼睛呈红色，其他部位呈浅灰色。

北非石鸡以种子、各种植物和昆虫为食。在春天，雌鸟在地上的巢里产 10 ～ 16 枚卵。它们的领地意识很强，经常叽叽喳喳地发出刺耳的叫声，宣告它们的“所有权”，这是它们的一个显著特征。目前，北非石鸡并未受到全球性的生存威胁。

不过，O’Reilly 封面上的许多动物正濒临灭绝，它们是自然界所剩无几的瑰宝。

封面插图是 Karen Montgomery 的作品，创作源于 *British Birds* 中的黑白版画。

O'REILLY®

Python深度学习入门：从零构建CNN和RNN

深度学习技术的发展如火如荼，这些知识正迅速成为机器学习从业者甚至许多软件开发工程师的“加分项”。深度学习是一个立体的领域，仅从数学层面或代码层面学习，难免以偏概全，无法融会贯通。

本书作者认为，理解深度学习和神经网络需要多种思维模型。因此，本书从数学、示意图、Python代码三个维度帮助你立体地理解每一个概念，带你领略深度学习领域的全貌，从内到外地理解构建神经网络的每一步。你将学到以下内容。

- **为理解深度学习的概念和原理构建多种思维模型。**
- **掌握嵌套函数、链式法则等数学概念。**
- **掌握学习率衰减、权重初始化、dropout等优化技巧。**
- **从零构建CNN和RNN等常见的神经网络架构。**
- **使用PyTorch实现神经网络。**

“无论你是否有经验，都可以借助这本书，从零开始理解和编码神经网络。”

——Pin-Yu Chen
IBM Research AI团队研究人员

塞思·韦德曼（Seth Weidman），SentiLink公司数据科学家。他曾在Facebook公司从事数据科学工作，并为多家企业开发了深度学习培训课程。塞思善于通过简单明了的方式解释复杂的概念。除了讲授课程，他还热衷于技术写作，并撰写了大量PyTorch教程。

DATA PROCESSING / DATA MODELING

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn

分类建议　计算机/深度学习/Python

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



扫码领取
随书代码资料


ISBN 978-7-115-55564-9

定价：79.00元